Hardware implementation of Elliptic Curve Cryptography based on Residue Number Systems

By

MohamadAli Mehrabi

A thesis submitted to Macquarie University
for the degree of Doctor of Philosophy
Department of Department of Computing
September 2020

MACQUARIE
University
SYDNEY · AUSTRALIA

© MohamadAli Mehrabi, 2020.

Typeset in $\LaTeX\,2_{\varepsilon}$.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

MohamadAli Mehrabi

Acknowledgements

I would like to thank my supervisors Dr Christophe Doche and Dr Alireza Jolfaei for their endless support and encouragement throughout my PhD candidature. Continuous encouragement, guidance, and feedback received from my supervisors helped me to achieve the milestones efficiently.

I am also thankful to my fellow group members for their friendly discussion and appreciation.

I wish to thank Naila Mukhtar for her help in the lab equipment preparation, ECC core testing, and side-channel power data gathering of the proposed ECC architectures.

I am thankful to the staff in the department of computing for their support. Finally, I would like to acknowledge Macquarie University for awarding me the Research Training Program Scholarship (RTP).

List of Publications

Published papers:

- Mohamad Ali Mehrabi, Christophe Doche, and Alireza Jolfaei, *Elliptic Curve Cryptography Point Multiplication Core for Hardware Security Module*, *IEEE Transaction on computers*, AUG. 2020.
- Mohamad Ali Mehrabi, Naila Mukhtar, and Alireza Jolfaei, *Side-channel power data analysis on RNS GLV ECC using Machine-Learning and Deep-Learning algorithms*, *ACM Transactions on Internet Technology*, SEP. 2020.
- Mohamad Ali Mehrabi and Christophe Doche, Low-Cost, Low-Power FPGA Implementation of ED25519 and CURVE25519 Point Multiplication, *Information*, vol. 10, no. 9, 285, 2019.
- Mohamad Ali Mehrabi, Improved sum of residues modular multiplication algorithm, *MDPI Journal of Cryptography*, vol. 3, no. 2, 14, 2019.
- Naila Mukhtar, Mohamad Ali Mehrabi, Yinan Kong, and Ashiq Anjum, Machine-Learning-Based Side-Channel Evaluation of Elliptic-Curve Cryptographic FPGA Processor, *MDPI journal of applied Sciences*, vol. 9, no. 1, 64, 2019.

Papers submitted for publication:

- Mohamad Ali Mehrabi and Alireza Jolfaei, *Efficient Cryptographic Hardware for Safety Message Verification in Internet of Connected Vehicles*, *ACM Transactions on Internet Technology*, Under Review.

Abstract

In today's technology, a sheer number of Internet of things applications use hardware security modules for secure communications. The widely used algorithms in security modules, for example, digital signatures and key agreement, are based upon elliptic curve cryptography (ECC). In many IoT applications, such as intelligent transportation systems and distributed control systems, thousands of safety messages need to be signed and verified within a very short time frame. Considerable research has been conducted in the design of fast elliptic curve arithmetic on finite fields using residue number systems (RNS). This thesis investigates fast hardware implementations for RNS elliptic curve cryptography (ECC) co-processors. Our focus is to speed up the ECC point multiplication operation by exploiting the properties of the residue number system (RNS). The RNS consists of independent and carry-free small-sized integer channels that make it suitable for long-integer arithmetic. By harnessing the RNS properties, hardware parallelism, and utilising different point multiplication algorithms, we designed a low-latency ECC point multiplication co-processor for the standard elliptic curves SECP256K1, ED25519, and Brainpool256r1 which are widely used in the industry. This thesis contributes to the field of hardware cryptography as follows: Two new architectures for RNS modular reduction are proposed. The first improvement is on the RNS Montgomery reduction algorithm in which its FPGA implementation utilises fewer hardware resources and is also much faster in terms of speed compared to the literature. In addition to the RNS modular reduction algorithm, a new modular reduction based on the sum of residues (SOR) is proposed. The SOR algorithm is highly parallelisable. Two variants of the SOR algorithm with different levels of parallelism are implemented on FPGA. Furthermore, the elliptic curve group law operations are optimised for parallel computation and are used in the design of an RNS ECC co-processor. This thesis analyses the security of RNS GLV ECC co-processors with respect to side-channel, power data analysis by making use of machine and deep learning algorithms. Finally, suitable countermeasures are proposed to make such co-processors immune to side-channel attacks.

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

List of symbols used in this thesis.

$\langle A \rangle_m$	$A \mod m$.
$\gcd(A, B)$	The greatest common divisor of A and B.
$\text{mods}(A, B)$	$\left\langle \langle A \rangle_B + \left\lfloor \frac{B}{2} \right\rfloor \right\rangle_B - \left\lfloor \frac{B}{2} \right\rfloor$.
$A \oplus B$	The bitwise XOR of the integers A and B.
$A \odot B$	The bitwise XNOR of the integers A and B.
$A \vee B$	The bitwise OR of the integers A and B.
$A \wedge B$	The bitwise AND of the integers A and B.
$\bar{A}$	The bitwise NOT of the integer A.
${A_i}^j$	The bit j of the integer A_i.
$a\|\|b$	Bit a concatenation to bit b.
$(A, B) \leftarrow (C, D)$	The value of C and D is loaded to registers A and B respectively at the same clock edge.
$\mathbb{R}$	Set of Real numbers.
$\mathbb{F}_p$	Prime Field.
$\mathbb{F}_{2^m}$	Binary Field.
$\#E$	The order of the elliptic curve E.
$A_{\mathcal{B}}$	The RNS presentation of A over the RNS base $\mathcal{B}$.
$(b_{n-1}b_{n-1}\ldots b_0)$	Binary representation of an n-bit integer B. ($b_i \in \{0, 1\}$).
'0'	Zero bit.
$\lceil u \rceil$	The function $ceil(u)$.
$\lfloor u \rfloor$	The function $floor(u)$.
$\bigoplus$	RNS addition operator.
$\mathbf{A}$	Matrix A.
$\mathbf{A}^T$	Transpose of Matrix A.
$\vec{X}$	Vector X.

$ReLu(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$ ...	Relu function.
$\otimes$	RNS multiplication operator.
$\circledcirc$	RNS squaring operator.
$\oplus$	RNS addition and modular reduction operator.
$\ominus$	RNS subtraction and modular reduction operator.
$\otimes$	RNS multiplication and modular reduction operator.
$\circledcirc$	RNS squaring and modular reduction operator.
●	RNS modular reduction operator.

List of Acronyms

1D-CNN	1-Dimensional Convolutional Neural Network
AES	Advanced Encryption Standard
ANN	Artificial Neural Networks
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
BE	Base Exchange
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPA	Carry Propagate Adder
CPU	Central Processing Unit
CSA	Carry Select Adder
CRT	Chinese Remainder Theorem
DAG	Directed Acyclic Graph
DBC	Double Base Chain
DBNS	Double Base Number System
DEMA	Differential Electro-Magnetic Analysis
DH	Diffie-Hellman Key Exchange
DL	Deep Learning
DNN	Deep Neural Network
DPA	Differential Power Analysis
DSCA	Differential Side-Channel Analysis
DSP	Digital Signal Processing
DT	Decision Tree
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem

ECDSA	Elliptic Curve Digital Signature Algorithm
ECPA	Elliptic Curve Point Addition
ECPD	Elliptic Curve Point Doubling
ECPM	Elliptic Curve Point Multiplication
ECPT	Elliptic Curve Point Tripling
ETSI	European Telecommunications Standards Institute
FA	Full Adder
FC	Fully Connected
FPGA	Field Programmable Gate Array
GAN	Generative Adversarial Network
GLV	Gallant-Lambert-Vanstone
HA	Half Adder
HDL	Hardware Description Language
ITS	Intelligent Transportation System
JSF	Joint Sparse Form
k-NN	k-Nearest Neighbours
LSTM	Long Short-Term Memory
LSB	Least Significant Bit
LUT	Look-Up Table
MESD	Multiple-Exponents, Single-Data Attack
ML	Machine Learning
MLP	Multi-Layer Perception
MMR	Montgomery Modular Reduction
MSB	Most Significant Bit
MR	Modular Reduction
MUX	Multiplexer
NAF	Non-Adjacent Form
NIST	National Institute of Standards and Technology
PCA	Principal Component analysis
POI	Point Of Interest
RAM	Random Access Memory
RF	Random Forest
RNN	Recurrent Neural Networks
RNS	Residue Number System

ROM	Read-Only Memory
RSA	Rivest-Shamir-Adleman
SECG	Standard for Efficient Cryptography Group
SEMA	Simple Electro-Magnetic Analysis
SEMD	Single Exponent Multiple Data Attack
SCA	Side-Channel Attacks
SNR	Signal to Noise Ration
SOR	Sum of Residues
SPA	Simple Power Analysis
SSCA	Simple Side-Channel Analysis
SVM	Support Vector Machine
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Very Large Scale Integration
ZEMD	Zero Exponent Multiple Data Attack

When theory and experiment agree, that is the time to be especially suspicious.

Niels Bohr

1 Introduction

In the recent decade, we have witnessed rapid developments in applications of public-key cryptography (PKC). The elliptic curve cryptography (ECC) especially has been at the centre of attention because it provides the same level of security as its counterparts, such as RSA and ElGamal, with smaller bit-size keys. Other than data security (confidentiality), PKC offers authentication, data integrity, and non-repudiation, which makes it suitable for a range of applications. Blockchain technology and Bitcoin cryptocurrency were introduced in 2009 using ECC as a core function to implement non-repudiation and authentication. Blockchain was found very interesting in business applications and became one of the pillars of FINTECH 3.0. Other than business and finance applications, Blockchain has been used in many different domains like digital voting and supply chain management. A wide range of cryptocurrencies is now available in the market, and they are becoming increasingly prevalent. Intelligent transportation systems (ITS) introduced in 2010, use ECC as their standard to sign and verify short communication and control car-to-car and car-to-infrastructures messages. The self-driving cars which first allowed on the roads for testing in 2015, use the same methods to authenticate communications. The IoT security and cloud computing security are other examples of new emerging applications that use elliptic curve cryptography as their underlying cryptosystem.

The security of the elliptic curve cryptosystems is based on the elliptic curve discrete logarithm problem (ECDLP). It is assumed that finding a random multiple of a known base point on the elliptic curve (EC) is computationally infeasible. A multiple of a point on the EC is calculated by a one-way function that is called "point multiplication" (or scalar multiplication). The point multiplication is a time-consuming process by nature. It uses the curve's algebraic formulas iteratively to calculate the new point. The core arithmetic primitive in the computation of a point coordinates is the modular reduction over the finite field characteristic, which is a complex arithmetic operation.

The latency of point multiplication imposes limitations in exploiting ECC in high-level applications. For instance, consider an ITS scenario. Cars and infrastructures must send digitally signed control and safety messages to each other and verify the messages they receive. A digital signature needs one point-multiplication, and verification needs two point-multiplications using an elliptic curve digital signature (ECDSA) scheme. By growing the number of cars on the road, verification of all incoming messages may not be possible; that may result in losing critical safety messages. A similar scenario applies to the self-driving cars technology, where decision-making relies on the verified messages received from roads infrastructures and other cars.

The latency of ECDSA is a determining factor of transactions speed in cryptocurrencies and verification of blocks in the Blockchain technology. The overall functionality of these applications can be limited by the latency of the cryptographic functions. Hence, this latency to be reduced as much as possible. The solution is the use of cryptographic co-processors. A co-processor is a hardware core that performs supplement functions of the primary processor (CPU).

Since the early days of computers, co-processors have been used to relieve the main CPU and accelerate processing times. They may be used for floating-point arithmetic (FPU), as graphics accelerators GPU), digital signal processing (DSP), or I/O interfacing (such as PCI and USB chips).

Cryptographic hardware has been an overwhelming research topic in recent years. Significant research has been done to reduce the latency of the elliptic curve point multiplication. There are different approaches to the problem. Many works in the literature, have tried to improve the elliptic curves group laws and propose algorithms to perform point multiplications in fewer iterations. Some works focused on improving modular arithmetic as the core function of the most public-key cryptosystems. Some other works concentrated on the implementation methods of cryptographic hardware, VLSI algorithms, and timing closure problems on either ASIC or FPGA devices.

An essential criterion the cryptographic co-processors must meet is physical security. A cryptographic algorithm may be mathematically secure. However, the hardware that runs this algorithm can leak the information. In 1996, Kocher found that cryptographic systems can be broken by monitoring and analysing side-channel information such as timing, power consumption, and electromagnetic radiation. Side-channel analysis attacks are a serious threat to the security of embedded devices, such as smart-cards, IoT devices, Blockchain, cryptocurrencies, etc. Recently, side-channel attacks on applications of ECC cryptosystems have been extensively studied. Matthews [1] showed that the security of smart cards can be compromised using low cost side-channel attacks. Wunan et al. [2] presented a side-channel attack case on Blockchain's ECDSA and acquired the private key. In [3], San Pedro et al. performed a successful side-channel attack on the hardware cryptocurrency wallets. A most recent research paper from Stanford university [4], showed that privacy focused cryptocurrencies like Zcach and Monero are vulnerable against side-channel attacks.

In a nutshell, there are numerous examples of how implementations of ECC algorithms resulted in significant vulnerabilities in the cryptographic software or hardware.

The application of machine-learning and deep-learning in the side-channel data analysis has made side-channel attacks increasingly powerful. It has been shown [5] that such techniques can efficiently deal with desynchronised traces and even attack masked implementations.

As side-channel attacks are becoming more powerful, research on finding effective countermeasures to dismiss the revealed vulnerabilities is essential. Otherwise, the security promised by cryptosystems, in theory, would be unreliable in the real world.

This thesis studies the hardware implementation of the ECC in the context of Residue Number Systems (RNS). RNS represents integers by their values modulo several pairwise co-prime integers named the moduli. RNS is an active area of research allowing faster public-key cryptography operations due to its inherent parallelism. The contribution of the thesis is in three areas.

First, I proposed efficient hardware implementations for RNS arithmetic. The proposed implementations are based on an improvement in the Montgomery modular reduction algorithm used in RNS, as it is a dominant method for RNS modular reduction. Our experiments show the hardware implementation of the new RNS Montgomery algorithm on FPGA is more efficient than the existing works in the literature. In addition to the improvement of the Montgomery modular reduction algorithm, a new modular reduction algorithm based on the sum of residues (SOR) is proposed. The motive of this design is that the SOR algorithm is highly parallel and it can efficiently reduce the calculation time. The proposed hardware architectures were implemented with different levels of parallelism and they have shown a good trade-off between cost and speed.

Second, taking advantage of parallel computing methods, I proposed a new architecture for hardware implementation of ECC group laws. I used different point multiplication algorithms to design a low-latency ECC co-processor. The proposed architectures were implemented on the FPGA and the obtained results confirm a reduction in ECC point multiplication latency.

Third, I analysed the side-channel leakage data of an RNS ECC co-processor using machine and deep learning algorithms. The experimental results confirm the method is secure from side-channel attacks.

The structure of this thesis is as follows:

Chapter 2 provides the background required for reading this thesis. The chapter starts with an introduction to Residue Number Systems (RNS) and arithmetic operations in the context of RNS. It then continues with a brief discussion on elliptic curves and point multiplication algorithms. Next, the FPGA hardware design is briefly outlined. Side-channel data analysis and side-channel attacks and countermeasures are reviewed. Machine-learning and deep learning algorithms, and their applications in side-channel data analysis are discussed.

Chapter 3 describes the hardware design of the ECC arithmetic primitives. This chapter details improvements on RNS Montgomery and SOR modular reduction algorithms. It also presents the implementation results.

Chapter 4 proposes hardware architectures for the ECC co-processor. The RTL design of the ECC co-processor is described. Implementation is performed for three different elliptic curves which are widely used in the industry, including

- The curve SECP256k1, which is used for Bitcoin, Ethereum, and Blockchain.
- The curve ED25519, that is widely used in numerous network security protocols, such as Transport Layer Security (TLS) and Secure Shell (SSH).
- The curve Brainpool256r1, which is the recommended curve in the ITS security standard.

Chapter 5 elaborates on side-channel power data gathering, pre-processing, and analysis. Various machine and deep learning algorithms are studied for the analysis of side-channel data. A successful attack to the GLV RNS ECC hardware is performed and efficient countermeasures are proposed.

Chapter 6 discusses the achievements made in the field and concludes the thesis, future works and research directions.

Appendix A lists the computer programs used for the simulation of our proposed algorithms, including the RNS Montgomery reduction algorithm, the SOR reduction algorithm, and tree-based and Lévia - Thériault's DBC algorithms.

Appendix B lists VHDL hardware implementations of the proposed modular reduction algorithms and ECC co-processor implementations.

2 Background

2.1 The residue number systems

The Residue number systems (RNS) were discovered by Svoboda and Valach in 1955 [6] and independently by Garner in 1959 [7], who were trying to apply this numbering system into the implementation of fast arithmetic and fault-tolerant computing. Carry-free propagation and channel-independent properties of RNS make them well suited for long integers arithmetic. The residue number system uses a base of co-prime moduli $\mathcal{B} = \{m_1, m_2, \cdots, m_N\}$, called the *RNS base*, to split an integer X into small n-bit integers $\{x_1, x_2, \cdots, x_N\}$ where x_i is the residue of X divided by m_i denoted as $x_i = X \mod m_i$ or simply $x_i = \langle X \rangle_{m_i}$. Each modulus is called an *RNS channel*. The integer X is then represented using N RNS channels.

$$RNS(X) = \{x_1, x_2, \cdots, x_N\}. \tag{2.1}$$

The range of the RNS, also known as *dynamic range*, is computed as:

$$M = \prod_{i=1}^{N} m_i. \tag{2.2}$$

2.1.1 The Chinese Remainder Theorem

The Chinese scholar Sun Tzu described a riddle in his book *"Arithmetical classic"* in the third century.

We have things of which we do not know the number
If we count them by threes, we have two left over
If we count them by fives, we have three left over
If we count them by sevens, we have two left over
How many things are there?

Sun Tzu gave a rule, *Great Generalisation*, for the solution of his puzzle. In 1247, another Chinese scientist, Qin Jiushao, generalised the Great Generalisation into what is called the

Chinese Remainder Theorem (CRT) today [8].
The CRT has a close relationship with the RNS. Reconstruction of the integer X, $0 \leq X < M$, from its RNS form $\{x_1, \ldots, x_N\}$ is possible using the CRT [9]:

$$X = \langle \sum_{i=1}^{N} \langle x_i \cdot M_i^{-1} \rangle_{m_i} M_i \rangle_M . \tag{2.3}$$

Where $M_i = \dfrac{M}{m_i}$, and M_i^{-1} is the multiplicative inverse of M_i. In other terms, $M_i \cdot M_i^{-1} \mod m_i = 1$.
Throughout this thesis, we assume that:

$$2^n > m_N > \cdots > m_1 > 2^{n-1}. \tag{2.4}$$

As a result, the dynamic range falls in the range $2^{N\cdot(n-1)} < M < 2^{N\cdot n}$.
Based on the Chinese remainder theorem, The RNS representation of an integer with in the RNS dynamic range is unique.

2.1.2 Arithmetic operations in RNS

RNS is a non-positional numbering system. Arithmetic operations in RNS are categorised into simple and complex operations. Addition, subtraction, and multiplication are simple operations that can be performed very efficiently in RNS. These operations are performed on each channel independently. However, operations like division, sign detection, comparison, and modular reduction are complex in RNS.
Suppose, X and Y are two l-bit integers presented in RNS form that is, $RNS(X) = \{x_1, x_2, \cdots, x_N\}$, and $RNS(Y) = \{y_1, y_2, \cdots, y_N\}$. The integers A, S, Z are the addition, subtraction, and multiplication results of X and Y, respectively. Then, the RNS are calculated as follows:

$$RNS(A) = \{\langle x_1 + y_1 \rangle_{m_1}, \langle x_2 + y_2 \rangle_{m_2}, \cdots, \langle x_N + y_N \rangle_{m_N}\}. \tag{2.5}$$

$$RNS(S) = \{\langle x_1 - y_1 \rangle_{m_1}, \langle x_2 - y_2 \rangle_{m_2}, \cdots, \langle x_N - y_N \rangle_{m_N}\}. \tag{2.6}$$

$$RNS(Z) = \{\langle x_1 \cdot y_1 \rangle_{m_1}, \langle x_2 \cdot y_2 \rangle_{m_2}, \cdots, \langle x_N \cdot y_N \rangle_{m_N}\}. \tag{2.7}$$

Considering that the multiplication result $Z = X \cdot Y$ is a $2l$-bit integer. The CRT enforces the condition $Z < M$. Otherwise, the N-tuple RNS set in (2.7) does not represent the correct integer Z. In other terms, for any operation in RNS, the result must be within the dynamic range i.e. $[0, M - 1]$.

Additive inverse in RNS

The additive inverse in RNS is defined by:

$$RNS(X) + RNS(\bar{X}) = 0. \tag{2.8}$$

This is applied to every individual RNS channel, that is,

$$\begin{aligned} \forall i \in \{1 \cdots N\}, x_i + \bar{x}_i &= 0, \\ \bar{x}_i &= m_i - x_i. \end{aligned} \tag{2.9}$$

Therefore, the additive inverse or negation of integer X (denoted as $-X$) in RNS can be written as

$$RNS(-X) = \{(m_1 - x_1), (x_2 - m_2), \cdots, (m_N - x_N)\}. \tag{2.10}$$

Multiplicative inverse in RNS

The multiplicative inverse of an RNS is defined by

$$RNS(X) \cdot RNS(X^{-1}) = RNS(1). \tag{2.11}$$

Similarly, this is applied to every individual RNS channel.

$$\begin{aligned} \forall i \in \{1, \cdots, N\}, x_i \cdot {x_i}^{-1} &= 1, \\ {x_i}^{-1} &= \langle x_i \rangle^{-1}{}_{m_i}. \end{aligned} \tag{2.12}$$

The multiplicative inverse of each RNS channel is the multiplicative inverse of that channel with respect to the corresponding RNS base modulus. In general, the multiplicative inverse of X_i exists only if x_i and m_i are relatively prime. In this case, the Fermat's little theorem can be used to determine the multiplicative inverse.

$$\langle {x_i}^{-1} \rangle_{m_i} = \langle {x_i}^{m_i - 2} \rangle_{m_i}. \tag{2.13}$$

The extended Euclidean algorithm is an extension to the Euclidean algorithm that computes the greatest common divisor (gcd) of two integers a and b, and also the coefficients of Bézout's identity which are integers x and y such that $ax+by = gcd(a, b)$. When a and b are co-primes, i.e. $gcd(a, b) = 1$ then $\langle ax + by \rangle_b = 1$. Hence, $\langle ax \rangle_b = 1$, or $a^{-1} \equiv x \mod b$. Algorithm 1 is a modification of the extended Euclidean algorithm [10] that calculates the multiplicative inverse of ${x_i}^{-1}$ using x_i and m_i as inputs.

Algorithm 1: Multiplicative inverse of a modulo b using Extended Euclidean algorithm.

```
Input: $a, b$ such that $a \in [1, b-1]$, $gcd(a, b) = 1$.
Output: $a^{-1} \mod b$.
1 $u \leftarrow a, v \leftarrow b$ ;
2 $x_1 \leftarrow 1, x_2 \leftarrow 0$ ;
3 while $u \neq 1$ do
4     $q \leftarrow \lfloor \frac{v}{u} \rfloor$ , $r \leftarrow v - qu$, $x \leftarrow x_2 - qx_1$;
5     $v \leftarrow u$, $u \leftarrow r$, $x_2 \leftarrow x_1$, $x_1 \leftarrow x$;
6 return $x_1 \mod b$.
```

Modular reduction in RNS

The Modular reduction is the core operation in public key cryptosystems where all calculations are performed in a finite field with characteristic p. Different methods are proposed in context of integer arithmetic including Barrett [11], Montgomery [12], and Interleaved [13] modular reduction. However, the first RNS modular reduction was proposed in 1995 by Karl and Reinhard Posch [14] based on the Montgomery reduction algorithm. Their proposed algorithm needed two RNS base extension operations. They used a floating-point computation for correction of the base extension in their architecture that was not compatible with the RNS representation. The base extension is a costly operation and limits the speed of the algorithm. In 1998, Bajard et al. [15] introduced a new Montgomery RNS reduction architecture using a mixed-radix system (MRS) [9] representation for base extensions. Due to the recursive nature of MRS, this method is hard to implement in hardware. Based on Shenoy and Kumaresan's work in [16], Bajard et al. proposed a Montgomery RNS modular reduction algorithm using residue recovery for the base extension [17]. In 2000, the floating-point approach of [14] was improved by Kawamura et al. [18] by introducing the cox-rower architecture. In 2014, Bajard and Merkiche [19] proposed an improvement in the cox-rower architecture by introducing a second level of Montgomery reduction within each RNS unit. The cox-rower architecture is well adapted for hardware implementation.

Let us start with the CRT. Equation (2.3) can be written as

$$X = \sum_{i=1}^{N} \gamma_i M_i - \alpha M, \tag{2.14}$$

where $\gamma_i = x_i {M_i}^{-1}$ and α is an integer. By dividing both sides of (2.14) by M, we obtain

$$\frac{X}{M} = \sum_{i=1}^{N} \gamma_i \frac{M_i}{M} - \alpha = \sum_{i=1}^{N} \frac{\gamma_i}{m_i} - \alpha. \tag{2.15}$$

Since $\frac{X}{M} < 1$ and $\frac{\gamma_i}{m_i} < 1$, it follows that

$$\alpha = \left\lfloor \sum_{i=1}^{N} \frac{\gamma_i}{m_i} \right\rfloor, \; 0 \le \alpha < N. \tag{2.16}$$

Hardware implementation of α has been discussed in [18]. It is shown that choosing proper integer constants q and Δ and enforcing boundary condition of (2.17), α can be calculated using (2.18).

$$0 \le X < (1 - \Delta)M. \tag{2.17}$$

$$\alpha = \left\lfloor \frac{1}{2^q} \left(\sum_{i=1}^{N} \left\lfloor \frac{\gamma_i}{2^{n-q}} \right\rfloor + 2^q.\Delta \right) \right\rfloor. \tag{2.18}$$

Algorithm 2 is used to calculate the coefficient α.

Algorithm 2: Calculation of α

```
Input: $\{\gamma_1, \dots \gamma_N\}$ where, $\gamma_i = \langle x_i \cdot {M_i}^{-1} \rangle_{m_i}$ , $i \in [1, N]$.
Required: $q$, $\Delta$.
Output: $\alpha$.
1 $A \leftarrow 2^q \cdot \Delta$ ;
2 for $i = 1$ to $N$ do
3     $A \leftarrow A + \left\lfloor \frac{\gamma_i}{2^{n-q}} \right\rfloor$;
4 end
5 return $\alpha \leftarrow \left\lfloor \frac{A}{2^q} \right\rfloor$ ;
```

RNS Base exchange

RNS base exchange is the critical operation in the RNS Montgomery reduction algorithm. Consider two RNS bases $\mathcal{K} = \{k_1, \dots, k_{N_1}\}$ and $Q = \{q_1, \dots, q_{N_2}\}$. Given an integer X in

Algorithm 3: RNS Base Exchange from $\mathcal{K}$ to $\mathcal{Q}$

```
**Required:** Base $\mathcal{K}$, Base $\mathcal{Q}$.
**Pre-compute:** $K = \prod_{i=1}^{N_1} k_i$, $Q = \prod_{i=1}^{N_2} q_i$, $K_i = \frac{K}{k_i}$, $Q_i = \frac{Q}{q_i}$, $U_{ij} = \langle K_i \rangle_{q_j}$ for $i = 1$ to $N_1$ and
    $j = 1$ to $N_2$
**Input:** $X_{\mathcal{K}}$.
**Output:** $X_{\mathcal{Q}}$.
1 $\gamma_i = \langle x_{k_i} K_i^{-1} \rangle_{k_i}$ ;
2 Compute $\alpha$ from Algorithm 2;
3 $V_{ij} = \left\langle \gamma_i U_{ij} \right\rangle_{q_j}$, $L_i = \langle -\alpha K \rangle_{q_i}$ ;
4 $x_{q_i} = \left\langle \sum_{j=1}^{N_2} V_{ji} + L_i \right\rangle_{q_i}$ ;
```

Algorithm 4: Montgomery modular reduction

```
**Required:** $\mathcal{K}, \mathcal{Q}, p_{\mathcal{K}}, p^{-1}{}_{\mathcal{Q}}, Q^{-1}{}_{\mathcal{K}}$.
**Input:** $X, p$.
**Output:** $Z = XR^{-1} \mod p$.
1 $s = X(-p)^{-1} \mod R$ ;
2 $t = X + s.p$ ;
3 $Z = tR^{-1}$;
```

base $\mathcal{K}$, that is, $X_{\mathcal{K}} = \{x_{k_1}, \ldots, x_{k_{N_1}}\}$, finding the RNS of X in base $\mathcal{Q}$ from its value in $\mathcal{K}$ is called *"base exchange"* from $\mathcal{K}$ to $\mathcal{Q}$. The base exchange operation is given in Algorithm 3.

The original Montgomery Modular reduction [12] in the context of integer is shown in Algorithm 4. The coefficient R is co-prime with the modulus p and chosen such that $R > 4p$. The RNS Montgomery reduction (Algorithm 5) is derived directly from the original Montgomery reduction algorithm (Algorithm 4). Here, R is replaced by $Q = \prod_{i=1}^{N_2} q_i$. In most cases, $N_1 = N_2 = \frac{N}{2}$, $\mathcal{K} = \{m_1, \ldots, m_{\frac{N}{2}}\}$ and $\mathcal{Q} = \{m_{\frac{N}{2}+1}, \ldots, m_N\}$. Thus, from (2.4) it follows that $Q > K$. Line 4 in Algorithm 4 is performed in base $\mathcal{Q}$. Since t is a multiple of Q, it is always zero in base $\mathcal{Q}$ [17]. Therefore, it is calculated in base $\mathcal{K}$ only. The computation in line 6 is to multiply Q by t. That can be carried out in base $\mathcal{K}$ but not in base $\mathcal{Q}$, since Q^{-1} does not exist in base $\mathcal{Q}$. Figure 2.1 shows the data flow diagram in the RNS Montgomery reduction algorithm.

Algorithm 5: RNS Montgomery modular reduction

```
**Required:** $\mathcal{K}, \mathcal{Q}, p_{\mathcal{K}}, p^{-1}{}_{\mathcal{Q}}, Q^{-1}{}_{\mathcal{K}}$.
**Input:** $X_{\mathcal{K} \cup \mathcal{Q}}$.
**Output:** $Z_{\mathcal{K} \cup \mathcal{Q}} = X \cdot Q^{-1} \mod p$ in base $\mathcal{K} \cup \mathcal{Q}$.
1 $s_{\mathcal{Q}} = X_{\mathcal{Q}} \cdot \langle -p^{-1} \rangle_{\mathcal{Q}}$ ;
2 $s_{\mathcal{K}} \leftarrow BaseExchange(s_{\mathcal{Q}})$ ;
3 $t_{\mathcal{K}} = s_{\mathcal{K}} + X_{\mathcal{K}}$;
4 $z_{\mathcal{K}} = t_{\mathcal{K}} \cdot Q^{-1}{}_{\mathcal{K}}$ ;
5 $z_{\mathcal{Q}} \leftarrow BaseExchange(z_{\mathcal{K}})$ ;
```

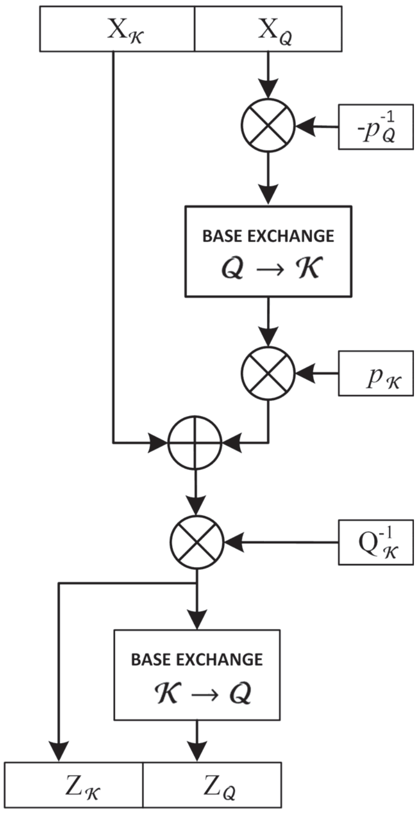


Figure 2.1: Flow diagram of RNS Montgomery reduction algorithm

The main advantage of the RNS Montgomery reduction algorithm is its efficiency in using computing resources. In this algorithm, half of the RNS channels are involved at a time. Thus, the hardware implementation is very area-efficient.

2.2 Elliptic Curve Cryptography

In the mid-1980s, Victor Miller from IBM and Neil Koblitz from the University of Washington independently introduced the concept of elliptic curve cryptography (ECC) [20], [21]. The ECC scheme provides a higher level of security per bit than other public-key cryptosystems. However, ECC also has a limitation. It requires more computations than its counterparts RSA or ElGamal. ECC is a public-key cryptosystem which is basically derived from the algebraic construction of elliptic curves over the finite fields. ECC has many advantages compared to other cryptographic schemes such as RSA, and ElGamal. One of the major advantages is that it can provide same level of protection offered by other cryptography schemes with keys of smaller size. For example, a 160-bit key ECC system provides same level of security as by RSA with 1024-bit key. Likewise, ECC with 224-bit and 256-bit keys provides same degree of protection provided by RSA with 2048-bit, and 3072-bit keys, respectively. ECC operates in smaller groups with smaller keys than other traditional public-key cryptosystems and although it requires more operations, overall encryption/decryption and signature need less hardware or software resources for implementation.

2.2.1 Elliptic Curves

Generally speaking, elliptic curves are "curves of genus one having a specified base point " [22]. The elliptic curve E over field $\mathbb{F}$, denoted by $E/\mathbb{F}$, is defined by Weierstraß equation [10]:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6. \tag{2.19}$$

where $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}$ and $\Delta \neq 0$, where Δ is the discriminant of E defined as:

$$\begin{aligned} \Delta &= -d_2{}^2d_8 - 8d_4{}^3 - 27d_6{}^2 + 9d_2d_4d_6 \\ d_2 &= a_1{}^2 + 4a_2 \\ d_4 &= 2a_4 + a_1a_3 \\ d_6 &= a_3{}^2 + 4a_6 \\ d_8 &= a_1{}^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3{}^2 - a_4{}^2. \end{aligned} \tag{2.20}$$

If K is any extension of Field $\mathbb{F}$ then the set of K-rational points on curve $E(K)$ is

$$\begin{aligned} &E(K) = \{(x, y) \in K \times K : \\ &\{y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\} \cup \{O\}. \end{aligned} \tag{2.21}$$

where O is the point at infinity. The curve E is defined over $\mathbb{F}$ when the coefficients a_1, a_2, a_3, a_4, and a_6 of its defining equation are elements of $\mathbb{F}$. If E is defined over a field, then it is also defined over any extension of that field. The condition $\Delta \neq 0$ ensures there are no points on the elliptic curve at which the curve has two or more distinct tangent lines. Such a curve is called none-singular. The K-rational points on E are the points (x, y) that satisfy the equation of the curve and whose coordinates $x, y \in K$. The point at infinity is considered a K-rational point for all extension fields K of $\mathbb{F}$.

The Weiersraß equation (2.19) can be mapped to a new form. If the characteristics of $\mathbb{F}$ is not 2 or 3 then using the mapping

$$(x, y) \rightarrow \left(\frac{x - 3a_1{}^2 - 12a_2}{36}, \frac{y - 3a_1x}{216} - \frac{a_1{}^3 - 4a_1a_2 - 12a_3}{24} \right) \tag{2.22}$$

the elliptic curve E is transformed to the short Weierstraß form

$$y^2 = x^3 + ax + b \tag{2.23}$$

where $a, b \in \mathbb{F}$ and the discriminant of the curve is $\Delta = -16(4a^3 + 27b^2)$.

2.2.2 The order of an Elliptic Curve

Let E be an elliptic curve defined over finite field $\mathbb{F}_p$. The number of the points on the $E(\mathbb{F}_p)$ denoted by $\#E(\mathbb{F}_p)$ is called the order of the curve E over the finite field $\mathbb{F}_p$. The Schoof algorithm [23] is used to find the order of E. Based on Hasse theorem [24], the order of an elliptic curve over the field $\mathbb{F}_p$ is within the range $p + 1 - \sqrt{p} \leqslant \#E(\mathbb{F}_p) \leqslant p + 1 + \sqrt{p}$. If $\#E(\mathbb{F}_p) = p + 1 - t$, then t is called the trace of the curve E.

2.2.3 Isomorphism and twist of an Elliptic Curve

Morphisms express algebraic relationships between elliptic curves. An isomorphism is a morphism of degree one. Essentially, an isomorphism is a change of coordinate system. For example, consider the curve $E : y^2 + y = x^3$ defined over $\mathbb{Q}$. There is an isomorphic map $(x, y) \rightarrow (2^2 3^3 x, 2^2 3^3 (2y + 1))$ from E to the curve $E' : y^2 = x^3 + 11664$ [25]. If two curves E and E' are isomorphic over K, then their groups $E(K)$, $E'(K)$ of K-rational points are also isomorphic. However, the converse is not true [10].

Now, consider the two curves E_1 and E_2 defined over K such that there is an isomorphism $E_1 \rightarrow E_2$ defined over $\overline{K}$, but not over K. In this case, curve E_2 is a *twist* of E_1. A twist of an elliptic curve E/K is another elliptic curve which is isomorphic to E over a new field $\overline{K}$. For instance, consider the curve $E/\mathbb{F}_p : y^2 = x^3 + ax + b$, The quadratic twist of E by a quadratic non-residue $D \in \mathbb{F}_p$ is $E_D : Dy^2 = x^3 + ax + b$. A quadratic non-residue means that one cannot find k such that $k^2 = D \mod p$. The curve $E_D : Dy^2 = x^3 + ax + b$ is isomorphic to $E_D : y^2 = x^3 + D^2 ax + D^3 b$ with the map $x = \frac{x}{D}$ and $y = \frac{y}{D\sqrt{D}}$. Evidently, the new y does not belong to $\mathbb{F}_p$ (because $\sqrt{D}$ is not in $\mathbb{F}_p$) but to the quadratic extension of $\mathbb{F}_p$. Thus, a point on E is mapped to another point on E_D but does not have coordinates in $\mathbb{F}_p$; which explains why E and E_D do not have the same number of points in $\mathbb{F}_p$.

2.2.4 Endomorphism of Elliptic Curves

An algebraic closure of a field K is a minimal algebraically closed field extension of K. The set of all points on the curve E that fall in any finite extension field of K is also denoted by E. An endomorphism ϕ is a map $\phi : E \rightarrow E$ such that [10]

$$\begin{aligned} \phi(P) &= (g(P), h(P)), \\ \phi(O) &= O. \end{aligned} \tag{2.24}$$

where h, g are rational functions i.e. their coefficients lie in K. The endomorphism ring of the elliptic curve E is the ring of all endomorphisms of E — including those defined over extensions of the base field of E.
The monic polynomial of the least degree such that $f(\phi) = 0$ is called characteristic polynomial of endomorphism ϕ.
For example, consider the curve $E : y^2 = x^3 + 7$ is defined over the finite field $\mathbb{F}_p$, where $p \equiv 1 \mod 3$. Let $\beta \in \mathbb{F}_p$ is an element of order 3. Then the map ϕ is defined as

$$\phi : (x, y) \rightarrow (\beta x, y), \tag{2.25}$$

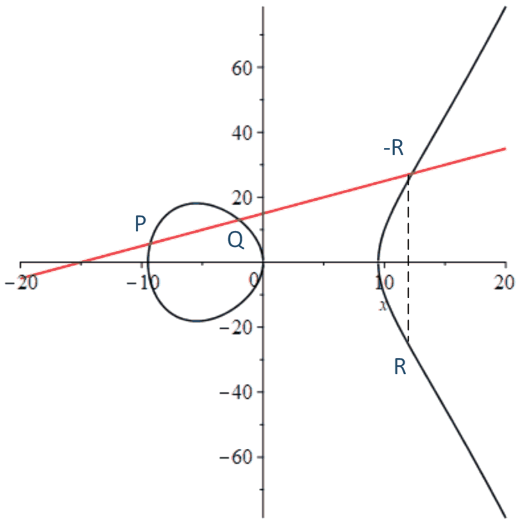
is an endomorphism of E over $\mathbb{F}_p$. The characteristic polynomial of ϕ is $X^2 + X + 1$. Suppose that $\langle P \rangle$, $P \in \mathbb{F}_p$ is the only subgroup of order n in $\mathbb{F}_p$; that is when n divides $\#E(\mathbb{F}_p)$ and n^2 does not divide $\#E(\mathbb{F}_p)$. Then $\phi(p) \in \mathbb{F}_p$ that results $\phi(p) \in \langle P \rangle$. If $\phi(P) \neq O$, then,

$$\phi(P) = \lambda P, \ \exists \lambda \in [1, n - 1]. \tag{2.26}$$

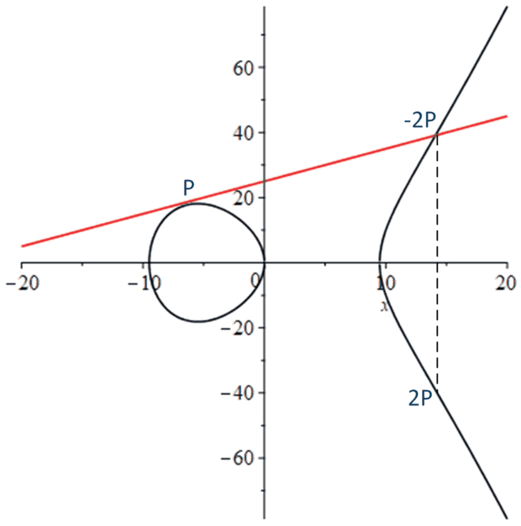
The λ is the root of characteristic polynomial of ϕ modulo n, i.e. $\lambda^2 + \lambda + 1 \equiv 0 \mod n$.

2.2.5 Group law on elliptic curves

Let $P(x_1, y_1)$ and $Q(x_2, y_2)$ are two point on the curve $E(K)$. The sum R of P and Q is geometrically defined as follows. First draw a line from point P to point Q. Then R is the reflection of the intersect point on the x-axis as depicted of figure 2.2a. Similarly, if we draw a tangent line to the elliptic curve at point P, the reflection of the intersect point is the point $2P$. as depicted in figure 2.2b. The algebraic formulae of elliptic curve point addition (ECPA) and elliptic curve point doubling (ECPD), can be derived from their geometrical definition. Considering the short



(a) Point addition on the elliptic curve



(b) Point doubling on the elliptic curve

Figure 2.2: Geometrically illustrated group law on elliptic curves on $\mathbb{R}$.

Weierstraß form elliptic curve (2.23), then for all $P(x_1, y_1), Q(x_2, y_2), R(x_3, y_3) \in E(K)$ it can be concluded that

- For $P = (x_1, y_1)$, $(x_1 + y_1) + (x_1, -y_1) = \mathcal{O}$. Then $-P = (x_1, -y_1)$.
- $P + \mathcal{O} = \mathcal{O} + P = P$.
- If $R(x_3, y_3) = P(x_1, y_1) + Q(x_2, y_2)$, then the point R coordinates can be calculated using

$$\begin{aligned} x_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right) - x_1 - x_2, \\ y_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1. \end{aligned} \tag{2.27}$$

- If $P \neq -P$, then $2P = (x_3, y_3)$, where

$$\begin{aligned} x_3 &= \frac{3{x_1}^2 + a}{2y_1} - 2x_1, \\ y_3 &= \left(\frac{3{x_1}^2 + a}{2y_1}\right)(x_1 - x_3) - y_1. \end{aligned} \tag{2.28}$$

It has to be noted that all operations are within the field $\mathbb{F}$.

2.2.6 Coordinate systems and group law

The point addition and point doubling formulae presentation in affine coordinates (2.27) and (2.28), require a one field inversion and several field multiplications. A field inversion is significantly more expensive than a field multiplication. That is why it is advantageous to represent points using projective coordinates. Projective coordinates are lines through the origin of a two-dimensional vector space. This space is called the projective space. A line in a projective space is given as a triple of (X, Y, Z), where $X, Y, Z \in \mathbb{F}$ and not all zero. The inverse of Z will exist for all nonzero $Z \in \mathbb{F}$. Two lines (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2) are equivalent if there exists a nonzero $\lambda \in \mathbb{F}$ such that $X_1 = \lambda^c X_2$, $Y_1 = \lambda^d Y_2$, $Z_1 = \lambda Z_2$. $(c, d$ are integers). Particularly, if $Z \neq 0$, then $(X/Z^c, Y/Z^d, 1)$ is a representative of the projective point (X, Y, Z) equivalent to the affine coordinate $(x = X/Z^c, y = Y/Z^d)$. Thus, we have a one-to-one correspondence between the set of projective points. The projective point $(X, Y, 0)$ is called the point at infinity. This point maps to the affine point O. In this way, the point at infinity (O) have clearly been defined as a point in the projective plane.

Standard Projective coordinates

Let $c = d = 1$, then the short Weierstraß elliptic curve equation (2.23) is transformed to

$$E : Y^2 Z = X^3 + aXZ^2 + bZ^3. \tag{2.29}$$

The only point on the projective line at infinity that also lies on E is $(0, 1, 0)$. This projective point corresponds to the point O in standard projective coordinates.

Jacobian coordinate

In the original definition of a projective space, the power d of λ is always 1. Later, a weighted variant was introduced to have even more efficient coordinate systems.

Let $c = 2$, $d = 3$. The point (X, Y, Z) in Jacobian coordinate is corresponding to the affine point $(X/Z^2, Y/Z^3)$. The short Weierstraß elliptic curve equation (2.23) is transformed to

$$E : Y^2 = X^3 + aXZ^4 + bZ^6. \tag{2.30}$$

The point at infinity is $(1, 1, 0)$ in Jacobian coordinates. Suppose $P = (X_1, Y_1, Z_1)$ and $2P = (X_3, Y_3, Z_3)$ are points on the curve E in Jacobian coordinates. The point doubling in Jacobian coordinates can be directly obtained from (2.27) by substitution of $x_1 = \frac{X_1}{{Z_1}^2}$ and $y_1 = \frac{Y_1}{{Z_1}^3}$.

$$\begin{aligned} X_3 &= A^2 - 2B \\ Y_3 &= A(B - X_3) - 8{Y_1}^4 \\ Z_3 &= Y_1 Z_1, \end{aligned} \tag{2.31}$$

where $A = \left(3{X_1}^2 + a{Z_1}^4\right)$, $B = 4X_1{Y_1}^2$.
Similarly, point addition formulae in Jacobian coordinates can be obtained from (2.28). if $R(X_3, Y_3, Z_3) = P(X_1, Y_1, Z_1) + Q(X_2, Y_2, Z_2)$, then

$$\begin{aligned} X_3 &= A^2 - B^3 - 2CB^2 \\ Y_3 &= A\Big(CB^2 - X_3\Big) - DB^3 \\ Z_3 &= Z_1 Z_2 B, \end{aligned} \tag{2.32}$$

where $C = X_1{Z_2}^2$, and $D = Y_1{Z_2}^3$.

Reorganising computations in the basic point-doubling and point-addition equations can lead to more efficient relations in terms of resource usage and speed. The Explicit Formulae Database (EFD) [26] is an online database for efficient point-arithmetic on elliptic curves.

2.2.7 Elliptic Curve Discrete Logarithm Problem

The Elliptic Curve Discrete Logarithm Problem (ECDLP) is the basis for the security of the elliptic curve cryptosystems.

Given an elliptic curve E over the prime field $\mathbb{F}_p$ denoted by $E(\mathbb{F}_p)$, a point $P \in E(\mathbb{F}_p)$ of order n, and the cyclic subgroup of $E(\mathbb{F}_p)$ generated by the point P, that is, $\langle P \rangle = \{O, P, 2P, \ldots, (n-1)P\}$, an integer $k \in [1, n-1]$ can be chosen randomly such that $Q = k \cdot P$. Here, the prime p, the curve E equation, the point P, and its order n are public domain parameters. Given the domain parameters and Q, the problem of determining k is named the Elliptic Curve Discrete Logarithm Problem [10]. The ECDLP is based on the intractability of the Discrete Logarithm Problem (DLP) on an elliptic curve. The point Q is easily computed with a given k using the one-way function $Q = k \cdot P$; that is, the elliptic curve point multiplication or scalar multiplication. Q is obtained by

$$Q = k \cdot P = \underbrace{P + P + \cdots + P}_{k \text{ times}}, \tag{2.33}$$

in which it is computationally difficult to determine k from known points Q and P. The fastest algorithm for solving ECDLP is Pollard's rho algorithm [27] which has a running time of $O(\sqrt{n})$, where n is the order of the elliptic curve E over the finite field $\mathbb{F}_p$.

2.2.8 Elliptic Curves Domain parameters

To implement elliptic curve cryptography, all parties must agree on all the ECDLP elements that define the elliptic curve; that is, the domain parameters of the scheme. The field is defined by the parameter p in the case of using prime finite field $\mathbb{F}_p$ or by parameters m, f in case of using binary finite field $\mathbb{F}_{2^m}$, where f is the irreducible polynomial modulus for polynomial basis. The elliptic curve is defined by the constants a and b used in the short Weierstraß equation. We normally work in a subgroup of order q. The aim is to find a point G of a large prime order q such that the co-factor $h = \frac{n}{q}$ is small, ideally equal to 1. It is important that q is prime otherwise, it is possible to reduce the ECDLP to a smaller problem using Silver-Pohlig-Hellman algorithm [10]. Since q is the size of a subgroup of $E(\mathbb{F}_p)$, based on the Lagrange's theorem [28], q divides n.

In conclusion, the domain parameters are (p, a, b, G, n, h) over a prime finite field and (m, f, a, b, G, n, h) over a binary field $\mathbb{F}_{2^m}$. The generation of domain parameters is done by standardisation bodies. Such domain parameters are commonly known as "standard curves". There are mainly three standard bodies which introduced their standard curves.

- NIST curves [29]:
  The National Institute of Standards and Technology (NIST) —that is a physical sciences laboratory and a non-regulatory agency of the United States Department of Commerce— introduced FIPS-186 standard including recommended elliptic curves specifications known as NIST recommended curves.

- SECG curves [30]:
  The Standards for Efficient Cryptography Group (SECG) is an international consortium founded by Certicom. The SECG develops commercial standards for efficient and interoperable cryptography. The SEC2 version 2.0 is the updated recommended domain parameters for elliptic curves.

- Brainpool curves [31]:
  The Brainpool curves are a new set of elliptic curve groups over finite prime fields for use in cryptographic applications. The parameters were generated in a pseudo-random way and have been verified to resist current crypto-analysis approaches.

2.2.9 Edwards curves[1]

Based on Euler and Gauss' works, Edwards introduced a normal form of elliptic curves in 2007 [33]. He generalised the curve as

$$y^2 + x^2 = a^2(1 + x^2y^2) \tag{2.34}$$

over the field K, where $a \in K$, such that $a^5 \neq a$. As Edwards stated in his paper, every curve of the form given in (2.34) is birationally equivalent to an elliptic curve in Weierstraß form. Two curves are considered birationally equivalent if their fields of rational functions are isomorphic. Bernstein et al. [34] generalised Edwards' original curves. For a fixed field K of odd characteristic and arbitrary integers $c, d \in K$ such that $cd(1 - dc^4) \neq 0$, they introduced the curves as

$$y^2 + x^2 = c^2(1 + dx^2y^2). \tag{2.35}$$

This definition covers "more than $1/4$ of all isomorphism classes of elliptic curves over a finite field". They showed that every elliptic curve over a non-binary field is birationally equivalent to a curve in Edwards form over an extension of the field and in many cases over the original field [34]. It can be easily verified that the neutral point (also called the point at infinity) O is $(0, c)$, and the inverse of point $P(x, y)$ is $(-x, y)$. In [35], Bernstein et al. introduced a generalisation of Edwards curves named *twisted Edwards curves*. These include more curves, including Edwards curves and every elliptic curve in Montgomery form [36]. As explained in [35], the curve name comes from the fact that the set of twisted Edwards curves is invariant under quadratic twists while a quadratic twist of an Edwards curve is not necessarily an Edwards curve. For a field K of odd characteristic, and nonzero elements $a, d \in K$, the twisted Edwards curve $E_{T,a,d}(K)$ is defined as

$$E_{T,a,d}(K) : ax^2 + y^2 = 1 + dx^2y^2. \tag{2.36}$$

If $a = 1$, then $E_{T,a,d}$ is an Edwards curve with $c = 1$. Moreover, $E_{T,a,d}$ is a quadratic twist of the Edwards curve $E_{O,1,d/a}$ with the map $(\overline{x}, \overline{y}) \rightarrow (x, y) = (\frac{\overline{x}}{\sqrt{a}}, \overline{y})$ over the field extension $K(\sqrt{a})$:

$$\overline{x}^2 + \overline{y}^2 = 1 + (d/a)\overline{x}^2\overline{y}^2 \tag{2.37}$$

Twisted Edwards curves and Montgomery curves are closely related. As shown in [35], every twisted Edwards curve $E_{T,a,d}$ on the Field K with $char(K) \neq 2$, is birationally equivalent to a Montgomery curve $E_{M,A,B} : Bv^2 = u^3 + Au^2 + u$ using the map

$$(x, y) \rightarrow (u, v) = \left(\frac{(1 + y)}{(1 - y)}, \frac{(1 + y)}{(1 - y)x}\right), \tag{2.38}$$

where $A = \dfrac{2(a + d)}{(a - d)}$, and $B = \dfrac{4}{(a - d)}$.

[1]This section is a part of published work [32]

If a is a square in K, then these curves are isomorphic over K itself. From the operation counts of the point arithmetic given in [26], it is easy to see that twisted Edwards curves outperform curves in Weierstraß form in terms of speed (despite the binary form of Edwards curve that is a bit slower than its Weierstraß counterpart [37]). However, twisted Edwards curves are appealing for another reason. Their group laws are unified and complete; that leads to safer implementations against certain types of attacks [35]. "Unified" means that the same addition equation works whether the points are the same or different. In other words, doubling and addition formulae are the same. "Complete" means that whatever the points are, the equation returns a correct result.

Point multiplication is fast and efficient on Montgomery curves. It efficiently uses differential point addition and point doubling [26] and uniform Montgomery ladder algorithm to perform a point multiplication [38]. The uniform Montgomery ladder algorithm is performed in constant time that makes its implementations robust to timing attacks. The CURVE25519 has been used in many software implementations since its introduction by Bernstein in [39]. It has also become a promising candidate for the Internet of Things (IoT) applications due to its 128-bit security level and efficient arithmetic.

The Edwards-curve Digital Signature Algorithm (EdDSA) is the most significant application of twisted Edwards curves. The ED25519 is a twisted Edwards curve used for EdDSA, where its parameters are defined as [40]

$$a = -1,$$

$$d = -\tfrac{121665}{121666}, \text{ and}$$

$$p = 2^{255} - 19.$$

The corresponding Montgomery curve of ED25519 is CURVE25519 that is defined as [39]

$$y^2 = x^3 + 486662x^2 + x. \tag{2.39}$$

2.2.10 Elliptic curves used in this research

In this research, our focus has been on the elliptic curves defined over 256-bit prime fields that provide 128-bit security. Specifically, the curves SECP256K1, ED25519, and Brainpool256r1 have been studied. The domain parameters of these curves are shown in Tables 2.1, 2.2, and 2.3.

Table 2.1: Curve SEC2P256K1 domain parameters

p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ (in decimal)
a = 0
b = 7
$G(x, y)$ =
(79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798,
547EF835 C3DAC4FD 97F8461A 14611DC9 C2774513 2DED8E54 5C1D54C7 2F046997)
n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
h = 1

Table 2.2: Curve Brainpool256r1 domain parameters

p = A9FB57DB A1EEA9BC 3E660A90 9D838D72 6E3BF623 D5262028 2013481D 1F6E5377
a = 7D5A0975 FC2C3057 EEF67530 417AFFE7 FB8055C1 26DC5C6C E94A4B44 F330B5D9
b = 26DC5C6C E94A4B44 F330B5D9 BBD77CBF 95841629 5CF7E1CE 6BCCDC18 FF8C07B6
$G(x, y)$ =
(8BD2AEB9 CB7E57CB 2C4B482F FC81B7AF B9DE27E1 E3BD23C2 3A4453BD 9ACE3262,
483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8)
n = A9FB57DB A1EEA9BC 3E660A90 9D838D71 8C397AA3 B561A6F7 901E0E82 974856A7
h = 1

Table 2.3: Curve ED25519 domain parameters

p = 7FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFED
$p = 2^{255} - 19$ (in decimal)
a = 7FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFEC
d = 52036CEE 2B6FFE73 8CC74079 7779E898 00700A4D 4141D8AB 75EB4DCA 135978A3
$G(x, y)$ =
(216936D3 CD6E53FE C0A4E231 FDD6DC5C 692CC760 9525A7B2 C9562D60 8F25D51A ,
66666666 66666666 66666666 66666666 66666666 66666666 66666666 66666658)
n = 10000000 00000000 00000000 00000000 14DEF9DE A2F79CD6 5812631A 5CF5D3ED

2.2.11 Elliptic Curve Diffie–Hellman

The elliptic curve Diffie–Hellman (ECDH) is a key exchange protocol that lets two parties have an elliptic curve public-private key pair to establish a shared key. Suppose, the two parties Alice and Bob want to share a secret over an insecure channel. They both need to agree on a domain parameter e.g. (p, a, b, G, n, h). Both Alice and Bob have to choose a private key randomly. Suppose that $k_A, k_B \in [1, n-1]$ are Alice's and Bob's private keys, respectively. Alice and Bob calculate their public keys using the base point G and their private keys. That is, $Q_A(x_A, y_A) = k_A \cdot G(x, y)$ and $Q_B(x_B, y_B) = k_B \cdot G(x, y)$, respectively. Now, both Alice and Bob have their private-public key pair i.e. (k_A, Q_A) and (k_B, Q_B). They exchange their public keys over the insecure channel. Now, Alice can compute $k_A \cdot Q_B$ and Bob can compute $k_B \cdot Q_A$. Therefore, the shared secret calculated by Alice and Bob is equal i.e. $k_A \cdot Q_B = k_B \cdot Q_A = k_A k_B \cdot G(x, y)$.

2.2.12 Elliptic Curve Point Multiplication algorithms

To compute the elliptic curve point multiplication $Q = k \cdot P$, several algorithms have been proposed some of which are Double-and-Add, window-NAF, and Montgomery ladder algorithms [41]. Some other variations of point multiplication algorithms are presented and discussed in [42], [43], [44], and [45]. In the following, we briefly introduce point multiplication algorithms used in this research.

Binary Double-and-Add algorithm

The classic binary Double-and-Add algorithm shown in Algorithm 6 [10] is a standard way to calculate equation (2.33). We assume that for an m-bit integer k, where $m = \lceil \log_2 k \rceil$, the binary representation of k is $(k_{m-1} k_{m-2} \cdots k_0)_2$, where $k_i \in \{0, 1\}$, $i \in [m-1, 0]$. Using

Algorithm 6: Binary Double and Add point multiplication algorithm

```
Input: $k, P$.
Output: $Q = k \cdot P$.
1  $Q = O$;
2  for $i = m - 2$ to $0$ do
3      $Q = 2P$;
4      if ( $k_i = 1$) then
5          $Q = Q + P$;
```

the Double-and-Add algorithm, m point-doublings and $\frac{m}{2}$ point-additions are required on average to perform a point multiplication. The double-and-add algorithm does not require pre-computation of points and extra registers to store the intermediate results. It is simple and has a small logic which makes it suitable for a low-cost RNS ECC hardware implementation. However, the double-and-add algorithm is vulnerable to side-channel attacks [46].

Montgomery ladder algorithm

The Montgomery ladder approach [41] computes the point multiplication in a fixed amount of time. This is particularly useful when timing or power consumption is exposed to side-channel attacks. The Algorithm 7 shows the Montgomery ladder method for computing point multiplication. An initial point-doubling is required to compute the value of $2P$. Then, P and $2P$ are saved as default values of R_0 and R_1 registers, respectively. One point-doubling and one point-addition are performed for every $m-1$ least significant bit (LSB) of the key k. Then, the value of the R_0 and R_1 registers are updated. If hardware resources are available, point-doubling and point-addition can be executed concurrently. The latency of the algorithm can be reduced to the latency of one point-doubling and $m - 1$ point-additions. Otherwise, the total latency is equal to the latency of m point-doublings and $m - 1$ point-additions.

Algorithm 7: Montgomery ladder algorithm

```
Input: $k, P$.
Output: $Q = k \cdot P$.
1  $R_0 = P$;
2  $R_1 = 2P$;
3  for $i = m - 2$ to $0$ do
4      if ( $k_i = 1$) then
5          $R_0 = R_0 + R_1$ ;
6          $R_1 = 2R_1$;
7      else if ($k_i = 0$) then
8          $R_1 = R_0 + R_1$;
9          $R_0 = 2R_0$;
10 return $R_0$.
```

NAF method

The Non-Adjacent-Form (NAF) representation of a positive m-bit integer k is defined as [10]

$$k = \sum_{i=0}^{l-1} k_i 2^i, \tag{2.40}$$

where $k_i \in \{-1, 0, 1\}$ and $k_{l-1} \neq 0$. The NAF representation of the integer k is denoted as NAF(k). Let us denote the length of NAF(k) by l which is at most one more than the length of its binary representation m. Any integer k has a unique NAF representation. The density is the number of non-zero digits divided by the total length. The average density of NAF representation is approximately $\frac{1}{3}$. The NAF of the integer k can be efficiently computed by letting $w = 2$ in Algorithm 8. The window-NAF method of length $w > 2$ denoted by NAF$_w$ is a generalisation of NAF. It requires $2^{w-2} - 1$ pre-computations at the cost of some extra computing resources. The latency of the NAF algorithm can be decreased by using the NAF$_w$ method presented in Algorithm 9, which processes w digits of k at a time. The density of NAF$_w$ is $\frac{1}{w+1}$. The $\text{mods}(a, b)$ operation used in Algorithm 8 is defined by

$$\text{mods}(a, b) = \left\langle \langle a \rangle_b + \left\lfloor \frac{b}{2} \right\rfloor \right\rangle_b - \left\lfloor \frac{b}{2} \right\rfloor. \tag{2.41}$$

Algorithm 8: Binary to NAF$_w$ conversion

```
  Input: $k, w$.
  Output: NAF representation of integer $k$.
1 $i = 0$;
2 while $k > 0$ do
3     if ($k \mod 2 = 1$) then
4         $k_i = 2^{w-1} - \text{mods}(k, 2^w)$;
5         $k = k - k_i$ ;
6     else
7         $k = \frac{k}{2}$ ;
8         $i = i + 1$;
9 return { $k_{i-1}, k_{i-2}, ..., k_0$ } .
```

Point multiplication using GLV method

In 2001, Gallant, Lambert, and Vanstone described a method — known as the GLV method— that the application of an endomorphism property if possible, can greatly speed-up the scalar multiplication operation.[47]. The curve E: SECP256K1 has an efficient endomorphism defined over $\mathbb{F}_p$. Here, $p \mod 3 = 1$. Let $\beta \in \mathbb{F}_p$ be an element of order 3. Then the map $\phi : E \to E$ defined by

$$\phi : (x, y) \to (\beta x, y), \quad \phi : \mathcal{O} \to \mathcal{O} \tag{2.42}$$

is an endomorphism of E defined over $\mathbb{F}_p$.
If $\phi(P) \neq \mathcal{O}$ then $\phi(P) = \lambda \cdot P$ where, λ is a root of the characteristic polynomial of ϕ satisfying $\lambda^2 + \lambda + 1 \equiv 0 \mod n$, ($n$ is the order of curve E) then for any point $P(x, y)$ on the curve:

Algorithm 9: NAF$_w$ point multiplication algorithm

```
   Input: $k, P, w$.
   Output: $Q = k \cdot P$.
 1 $Q = \mathcal{O}$ ;
 2 Use Algorithm 8 to compute NAF$_w(k)$= $\sum_{i=0}^{l-1} k_i 2^i$;
 3 Pre-compute $P_i = i \cdot P$ for $i \in \{1, 3, 5, \cdots, 2^{w-1} - 1\}$ ;
 4 for $i = l - 1$ to $0$ do
 5     Q = 2Q ;
 6     if ( $k_i > 0$) then
 7         $Q = Q + P_i$ ;
 8     else if ($k_i < 0$) then
 9         $Q = Q - P_i$ ;
10 return $Q$.
```

$$
\begin{aligned}
Q &= \phi(P(x, y)), \\
Q &= \lambda \cdot P(x, y) = (\beta x, y).
\end{aligned}
\tag{2.43}
$$

The numerical values of λ and β for curve SECP256K1 in hexadecimal notation are λ = 5363AD4CC05C30E0A5261C028812645A122E22EA20816678DF02967C1B23BD72, and β = 7AE96A2B657C07106E64479EAC3434E99CF0497512F58995C1396C28719501EE. Furthermore, for any $k \in [0, n-1]$ we can find k_1 and k_2 such that: $k = k_1 + \lambda k_2 \mod n$. Therefore,

$$
\begin{aligned}
k \cdot P &= k_1 \cdot P + k_2 \lambda \cdot P. \\
k \cdot P &= k_1 \cdot P + k_2 \cdot Q.
\end{aligned}
\tag{2.44}
$$

The value of k_1 and k_2 can be calculated using Algorithm 10. The length of k_1 and k_2, l is approximately half of length of k. The scalar multiplication $k \cdot P = k_1 \cdot P + k_2 \cdot Q$ will be calculated using Shamir's [48]. If k_1 and k_2 be used in their binary representation, then pre-computation of $P + Q$ is required and point multiplication can be done by l point doublings and $\frac{3l}{4}$ point additions on average. The computation is even more efficient when k_1 and k_2 are expressed in *Joint Sparse Form* (JSF) [48]. Suppose, $\begin{pmatrix} {k_1}^{l-1} & \cdots & {k_1}^0 \\ {k_2}^{l-1} & \cdots & {k_2}^0 \end{pmatrix}$ is the JSF representation of k_1 and k_2. Considering $R = k_1 \cdot P + k_2 \cdot Q$, Algorithm 11 calculates the simultaneous multiple point multiplication using Shamir's trick [10] and JSF of k_1 and k_2. Here, l point doublings and $\frac{l}{2}$ point additions are required on average. The points $(P + Q)$, $(P - Q)$, $-P$, and $-Q$ must be pre-computed. Since k_1 and k_2, are of half the bit-length of k, then the average latency of this method is approximately equal to half of the average latency of the Double-and-Add plus the latency of pre-computations that is, one modular multiplication (to calculate $Q = (\langle \beta x \rangle_p, y)$), one subtraction (to calculate $-P = (x, p - y)$, $-Q = (\beta x, p - y)$) and two point additions (to derive $P + Q$ and $P - Q$).

Algorithm 10: Calculation of k_1 and k_2

```
Input: Integer $k, n, \lambda$.
Output: Integers $k_1, k_2$ such that $k = (k_1 + k_2\lambda) \mod n$.
$w \leftarrow \lceil \log_2(n) \rceil$
$u \leftarrow \lambda, v \leftarrow n$
$t_1 \leftarrow 0, t_2 \leftarrow 1$
$s_1 \leftarrow 1, s_2 \leftarrow 0$
while $u > w$ do
    $q \leftarrow \lfloor v/u \rfloor, r \leftarrow (v - q.u), s \leftarrow (s_2 - q.s_1), t \leftarrow (t_2 - q.t_1)$.
    $v \leftarrow u, u \leftarrow r, s_2 \leftarrow s_1, s_1 \leftarrow s, t_2 \leftarrow t_1, t_1 \leftarrow t$.
$r_1 \leftarrow v$
$q \leftarrow \lfloor v/u \rfloor, r \leftarrow v - q.u, s \leftarrow s_2 - q.s_1, t \leftarrow t_2 - q.t_1$
$v \leftarrow u, u \leftarrow r, s_2 \leftarrow s_1, s_1 \leftarrow s, t_2 \leftarrow t_1, t_1 \leftarrow t$
$a_1 \leftarrow v$
$b_1 \leftarrow -s_2$
$q \leftarrow \lfloor v/u \rfloor, r \leftarrow v - q.u, s \leftarrow s_2 - q.s_1, t \leftarrow t_2 - q.t_1$
$v \leftarrow u, u \leftarrow r, s_2 \leftarrow s_1, s_1 \leftarrow s, t_2 \leftarrow t_1, t_1 \leftarrow t$
$r_2 \leftarrow v$
if $((r_1{}^2 + s^2) \geq (r_2{}^2 + s_2{}^2))$ then
    $a_2 \leftarrow r_1$
    $b_2 \leftarrow -s$
else
    $a_2 \leftarrow r_2$
    $b_2 \leftarrow -s_2$
$c_1 \leftarrow \lfloor b_2.k/n \rfloor$
$c_2 \leftarrow \lfloor -b_1.k/n \rfloor$
$k_1 = k - (c_1.a_1 + c_2.a_2)$
$k_2 = -(c_1.b_1 + c_2.b_2)$
return $k_1, k_2$
```

Algorithm 11: Simultaneous multiple point multiplication

```
Input: $P, Q$, JSF representation of $k_1$, $k_2$: $\begin{pmatrix} k_1{}^{l-1} & \cdots & k_1{}^0 \\ k_2{}^{l-1} & \cdots & k_2{}^0 \end{pmatrix}$
Output: $R = k_1 \cdot P + k_2 \cdot Q$.
Pre-compute: $-P$ , $-Q$, $(P + Q)$ , $(P - Q)$ ;
$R = \bar{O}$;
for $i = l - 1$ to $0$ do
    $R = 2R$;
    $R = R + (k_1{}^i \cdot P + k_2{}^i \cdot Q)$;
return $R$;
```

Algorithm 12: Greedy algorithm to find DBNS form of an integer

```
Input: Integer $k$.
Output: DBNS representation of $k$.
1 $t = k$;
2 $i = 0$;
3 while $t \neq 0$ do
4     Find $z = (-1)^{c_i} 2^{a_i} 3^{b_i}$ as the best approximation of $t$, that is, $t \approx z$;
5     $i = i + 1$;
6     $t \leftarrow t - z$;
7 return $(-1)^{c_0} 2^{a_0} 3^{b_0} + (-1)^{c_1} 2^{a_1} 3^{b_1} + \cdots$
```

Point multiplication using Double Base Number System (DBNS)

Double Base Number Systems were introduced by Dimitrov et al [49] and they are representations of a positive integer k in form of

$$k = \sum_{i=1}^{l-1} d_i 2^{a_i} 3^{b_i}, \tag{2.45}$$

where $d_i \in \{-1, 0, 1\}$. Such representations always exist and are not unique. Some of these representations, named canonic representations, which require a minimal number of additions of $\{2, 3\}$-integers, are of special interest. Canonic representations are sparse and finding them in a reasonable amount of time, especially for large integers, is challenging. To find a DNBS representation of an integer, the authors of [50] proposed a greedy approach as shown in Algorithm 12. This algorithm is efficient and returns a DBNS in a short amount of time, even for a large integer. However, computing a scalar multiplication based on the DBNS may not lead to a realistic implementation [51]. For example, if we attempt to compute $Q = 575543P$ using a DBNS representation obtained from a Greedy algorithm, that is, $575543 = 2^8 3^7 + 2^9 3^3 - 2^6 3^3 + 2^5 3^1 + 2^3 3^1 - 1$, which can be re-written as $((((2^{-1} 3^4 + 1)2^2 - 1)2^2 3^2 + 1)2^2 + 1)2^3 3^1 - 1$, then, we have to start with four point-tripling and one division by two (or one point-halving) to obtain $[2^{-1} 3^4]$. Point halving is only practical for curves of characteristic 2. Such an implementation requires extra hardware for point-halving and is not efficient. Therefore, when the two sequences of exponents are not decreasing simultaneously, the implementation of ECC scalar multiplication is not efficient in practice.

The concept of Double-Base Chains (DBC) is introduced by requiring the condition $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_{l-1}$ and $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_{l-1}$. The Greedy algorithm can be modified easily to compute such a DBC ensuring that the exponents a_i and b_i are decreasing [52].

Finding DBC based on Tree algorithm

The tree-based DBC search algorithm introduced in [52] is a generalisation of the binary/ternary method presented by Ciet in [53]. Considering that for a positive integer k, $\mathrm{mods}(k, 2) \in \{0, 1\}$ and $\mathrm{mods}(k, 3) \in \{-1, 0, 1\}$, if k is co-prime to 3, then 3 is a divisor of $k-1$ or $k+1$. Obviously, if k is odd, then $k - 1$ and $k + 1$ are even. Assuming that k is co-prime to 6, the tree based DBC search algorithm first builds a tree with $k - 1$ and $k + 1$ branches. After removing the powers of 2 and 3 from $k - 1$ and $k + 1$ this process will be reapplied to each branch, adding and subtracting 1, creating new branches until one of the branches will reach 1, leading to the

DBC expansion of k. This approach is costly especially for the integers in the range of ECC (160 to 512 bits). However, we can define a bound $\mathfrak{B}$ and keep only $\mathfrak{B}$ branch nodes at each level and eliminate other branches to speed up calculations.

The Tree algorithm considers only some special greedy chains. When it reaches an even node, it keeps on dividing by 2, prohibiting nontrivial additions. Similarly, when it reaches a multiple of 3 node, it keeps on dividing by 3. When it sees a number that is neither a multiple of 2 nor 3 it forms a branch by adding and subtracting one. This algorithm reduces the number of nodes but does not guaranty to find the best chain necessarily. Figure 2.3a illustrates the tree diagram of integer 575543 based on Algorithm 13. The result is the chain

$$575543 = ((((3^2 \cdot 2^2 + 1)3^3 \cdot 2^2 + 1)3 \cdot 2) - 1)3 \cdot 2^3 - 1. \tag{2.46}$$

For implementation purposes, we represent the DBC in the form of a chain of tuples. The pair $[2, 0]$ means one point-doubling, $[2, 1]$ means one point-doubling and one point-addition, $[3, 0]$ means one point-tripling and so on. Thus, the above example can be displayed as

$$\begin{aligned} 575543 =& [3,0][3,0][2,0][2,1][3,0][3,0][3,0][2,0][2,1][3,0] \\ & [2,-1][3,0][2,0][2,0][2,-1]. \end{aligned} \tag{2.47}$$

This chain consists of seven point-triplings, eight point-doublings and four point-additions (or subtraction).

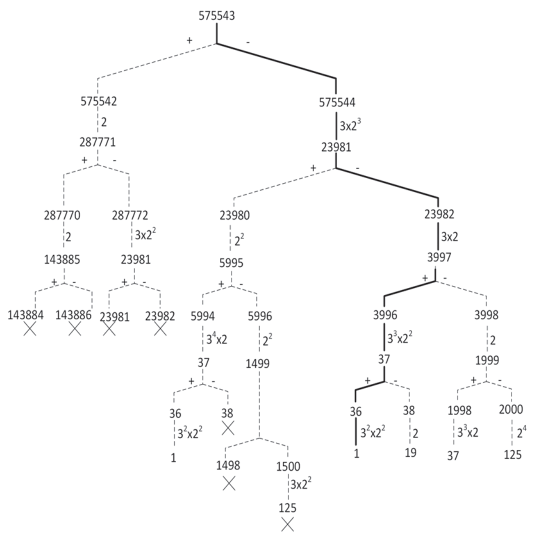
Algorithm 13: Tree-based DB-Chain search algorithm

Input: Integer k, Bound B.
Output: A DBC equivalent to k.

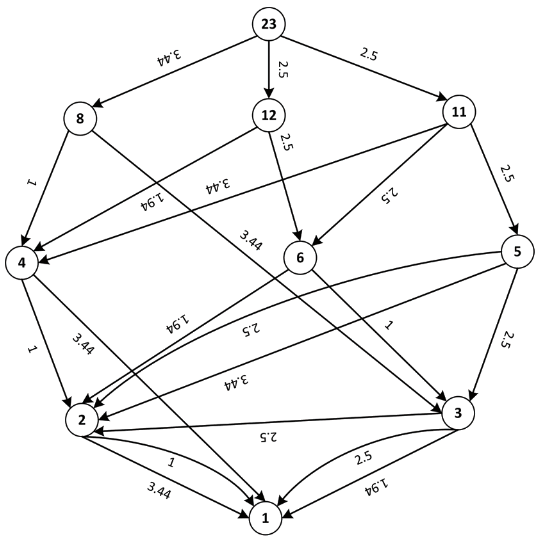
```
1 Set $t = f(k)$, $[f(k) = \frac{k}{2^{v_2(k)}2^{v_3(k)}}]$
2 Initialise binary tree $\mathcal{T}$ with root node $t$
3 while (a branch node is not equal to 1) do
4     for each branch node $m$ in $\mathcal{T}$ insert 2 Children do
5         Left child $\leftarrow f(m-1)$ ;
6         Right child $\leftarrow f(m+1)$ ;
7         Discard any redundant branch node ;
8         Discard all but the $B$ smallest branch node ;
9 return $\mathcal{T}$ ;
```

Finding DBC based on DAG method

Finding a canonic DBC by searching the shortest path in an explicit Directed Acyclic Graph (DAG) was introduced in [54]. An optimal length DBC algorithm based on the DAG method was proposed by Lévia and Thériault in [55] (we call it the L-T algorithm from now on). With the DAG method, every even node n produces two new nodes, one by dividing by 2, and the other $n-1$ or $n+1$ dividing by 3. Every odd node n will produce 3 nodes, $n+1$ divide by 2, $n-1$ divide by 2 and $n-1$ or $n+1$ divide by 3. Each possible intermediate result appears exactly once in the DAG but can appear at many different tree levels in the Tree-based algorithm that might not be considered. In the DAG method, the costs of doubling, tripling, and adding are considered to pick up the optimal low-cost chain. The disadvantage of this method is that a node can be produced multiple times in different positions. So, the number of nodes and the



(a) Using Tree algorithm to obtain DBC of 575543



(b) Using DAG to find DBC of 23

Figure 2.3: Finding canonic DBC using Tree and DAG algorithms

required memory for calculations is huge. Figure 2.3b illustrates DAG to find DBC of integer 23. The numbers on the edges of the DAG are the costs of operations to reach the destination node. In this example, normalised costs for Curve ED25519 are considered. It is assumed that a doubling cost is 1, a tripling cost is 1.94, and an addition or subtraction cost is 1.5. The output of the L-T algorithm for 575543 is

$$575543 = ((((3^2 \cdot 2^2 + 1)3^4 \cdot 2 + 1)2^2 + 1)3 \cdot 2^3) - 1. \tag{2.48}$$

2.3 Hardware design

In this section, the aspects of hardware design and optimisation will be discussed with a focus on Field Programmable Gate Arrays (FPGA). An FPGA is a type of integrated circuit that can be programmed for implementation of different algorithms. Modern FPGA devices consist of up to two million logic cells that can be configured to implement a variety of software algorithms. An FPGA provides significant cost advantages in comparison to an Application Specific Integration Circuit (ASIC) and offers the same level of performance in most cases. Another advantage of the FPGA compared to the ASIC is its ability to be dynamically reconfigured. This process, which is the same as loading a program in a processor, can affect a part or all of the resources available in the FPGA fabric. Hardware description languages (HDL) are mainly used to configure hardware on the FPGA. The two most common HDLs are VHDL and Verilog. Like a program written in any other language, an HDL program can be executed. Since HDL is used to model a Hardware, the term simulation is often used instead of execution, with the same meaning. An HDL program can be transformed with a synthesis tool into a netlist, that is, a detailed list of hardware primitives (e.g. gates and flip-flops) and connections between them. There are two different methodologies when designing an integrated circuit. The traditional method of hardware design is “Bottom-up” that is performed at the gate-level using the standard gates. This approach is nearly impossible to maintain with the increasing complexity of the design. Modern hardware systems usually consist of thousands of gates.

The traditional bottom-up designs must give way to new structural, hierarchical design methods. The desired design-style of all designers is the “Top-down” design. A top-down design method allows early testing, easy change of different technologies, a structured system design, and offers many other advantages. However, It is very difficult to follow a pure top-down design and in most cases does not yield an optimised result. Due to this fact, most designs are a mix of both methods, implementing some key elements of both design styles.

Hardware design abstraction levels

The hardware description may be performed at many different levels of abstraction. When considering the FPGA/ASIC design, three levels of abstraction are mainly identified.

- Behavioural level
- Register-Transfer Level (RTL)
- Gate level

Behavioural level

Behavioural modelling describes how the circuit should behave. At this level, a system is described by concurrent algorithms. The HDL behavioural model is widely used in the simulation of the design since the simulation just shows the functionality of the design and does not care about the hardware realisation. Some high-level synthesis tools can synthesise the behavioural models. The actual hardware implementation, in this case, will be decided by the synthesis tool.

RTL level

Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An RTL description has an explicit clock. All operations are scheduled to occur in specific clock cycles, but there are no detailed delays below the cycle level. The synthesis tools allow some freedom in this respect. A single global clock is not required but may be preferred. In addition, re-timing is a feature that allows operations to be re-scheduled across clock cycles.

Gate level

Within the gate or logic level, the characteristics of a system are described by logical links and their timing properties. The usable operations are predefined logic primitives (AND, OR, NOT gates).

2.3.1 FPGA architecture

Every FPGA chip is made up of a finite number of predefined resources with programmable interconnects to implement a reconfigurable digital circuit. It is important to have a basic understanding of the available resources in the FPGA fabric and how they interact to implement an application. FPGAs include a number of configurable logic blocks (CLB), a number of fixed-function logic blocks such as multipliers, and memory resources like embedded block RAM. The CLBs are the basic logic units of an FPGA for implementing sequential as well as combinatorial circuits [56]. As shown in Figure 2.4, each CLB element is connected to a switch matrix for the access to the general routing matrix. In Xilinx Virtex-7 FPGA family, a CLB element contains a pair of slices referred as SLICEM and SLICEL. Each Slice consists of

- Look-up table
- Flip-Flop
- Multiplexer
- Carry chain
- Wires

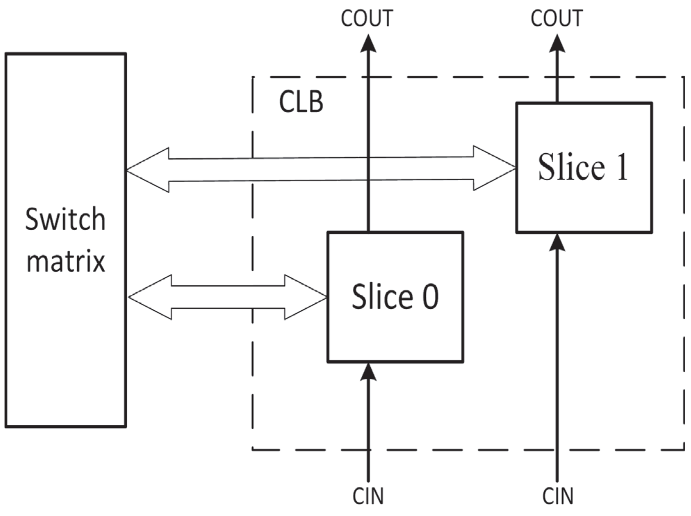


Figure 2.4: Configuration of a CLB in Virtex-7 family FPGAs (Courtesy of Xilinx).

Look-up table (LUT)

The LUT is the basic building block of an FPGA and is capable of implementing any combinational logic function of N Boolean inputs. This element is basically a truth table in which different combinations of the inputs implement different functions to produce the output values. The limit on the size of the truth table is N. For the general N-input LUT, the number of memory locations accessed by the table is 2^N which allows the table to implement 2^{N^N} functions. The Series-7 family of Xilinx FPGAs have LUTs with 6 inputs. Each slice consists of four 6-input LUTs.

Flip-Flop (FF)

This register element stores the result of the LUT. There are eight storage elements per slice. Four can be configured as either edge-triggered D-type flip-flops or level-sensitive latches. The basic structure of a flip-flop includes 'data input', 'clock input', 'clock enable', 'reset', and 'data output'. During normal operation, any value at the 'data input' port is latched and passed to the output on every pulse of the clock. The purpose of the 'clock enable' pin is to allow the flip-flop to hold a specific value for more than one clock pulse. New data inputs are only latched and passed to the 'data output' port when both 'clock' and 'clock enable' are set to one.

Multiplexers

There are two 7-input and one 8-input Multiplexers per slice. These multiplexers allow LUT combinations of up to four LUTs in a slice. The wide multiplexers can also be used to create general-purpose functions.

Carry Chain

In addition to function generators, dedicated fast look-ahead carry logic is provided at each slice to perform fast arithmetic addition and subtraction. A Virtex-7 family CLB has two separate carry chains that are cascadable to form a wider add/subtract logic.

Wires

Wires connect the elements within a slice together and to the switch matrix.

The function generators (LUTs) in SLICEMs can be implemented as a synchronous RAM resource called a distributed RAM element. Multiple LUTs in a SLICEM can be combined in various ways to store a larger amount of data. Maximum $256 \times 1-bit$ RAM can be implemented in one SLICEM. SLICEM LUTs can also be configured as a shift register without using the flip-flops available in a slice. Used in this way, each LUT can delay serial data from 1 to 32 clock cycles. Figure 2.5 shows the configuration of a SLICEM in the Virtex-7 family CLB. Advanced FPGA architectures include additional computational and data storage blocks that increase the computational density and efficiency of the device. These additional elements, which are discussed in the following are:

- DSP blocks for arithmetic computations.
- Embedded memories.
- Phase-locked loops (PLL).
- High-speed serial transceivers.

The combination of these elements provides FPGA with the flexibility to implement any software algorithm running on a processor and results in the contemporary FPGA architecture shown in Figure 2.6.

DSP

The DSP slice is the most complex computational block available in the modern FPGAs. A Block diagram of a DSP slice in Xilinx Series 7 family FPGAs is shown in Figure 2.7. The DSP block is an arithmetic logic unit (ALU) embedded into the fabric of the FPGA, it is composed of a chain of three different smaller blocks. The computational chain in the DSP includes an add/subtract unit connected to a 25×18 bits multiplier chained to a final add/subtract/accumulate unit. This chain allows a single DSP unit to implement functions in general form of $P = B \times (A + D) + C$ or $P+ = B \times (A + D)$.

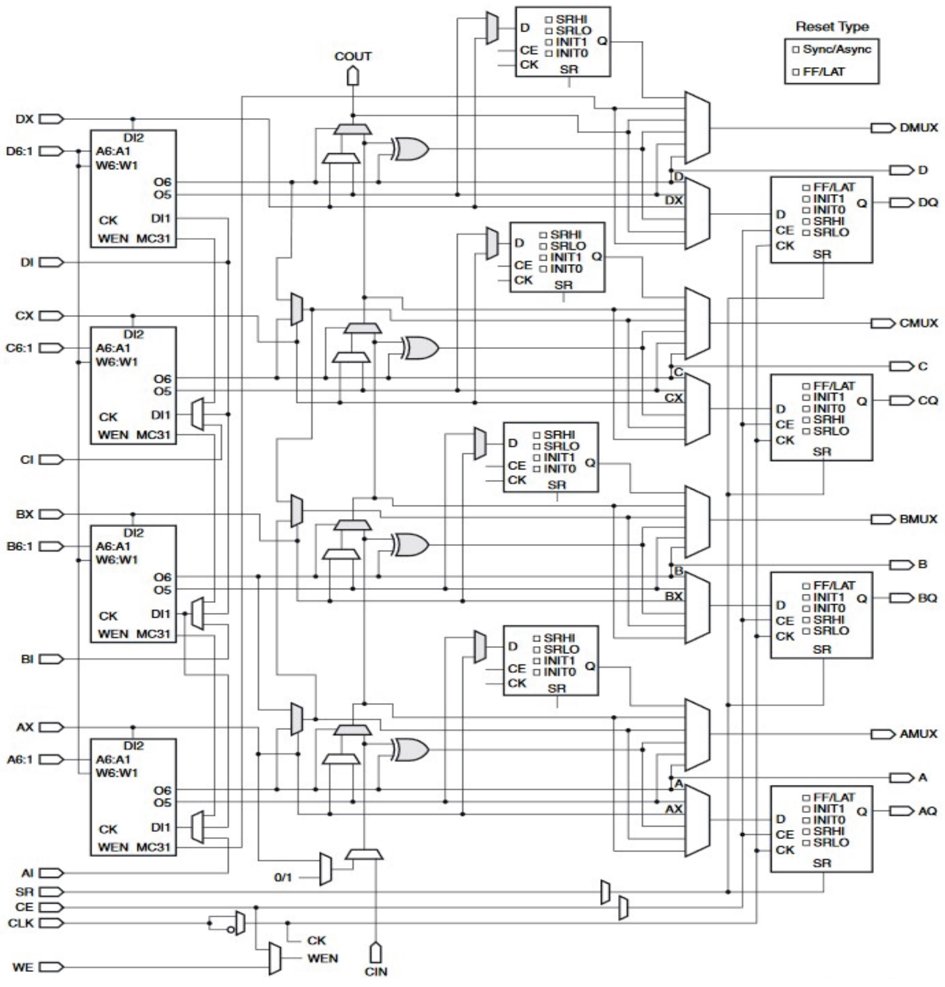
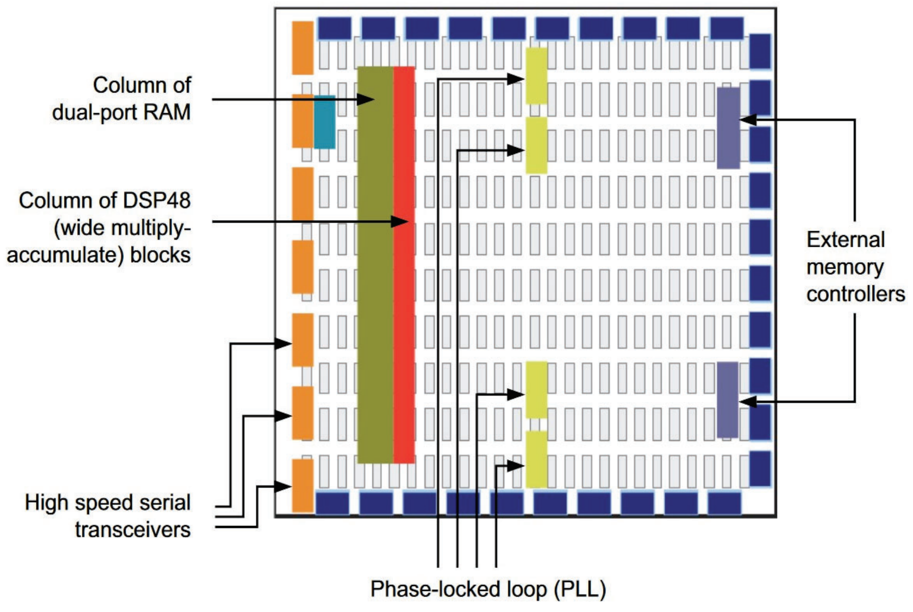


Figure 2.5: Architecture of a SLICEM in Virtex-7 family FPGAs (Courtesy of Xilinx).



Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
External memory controllers
Phase-locked loop (PLL)

Figure 2.6: Contemporary FPGA layout (Courtesy of Xilinx).

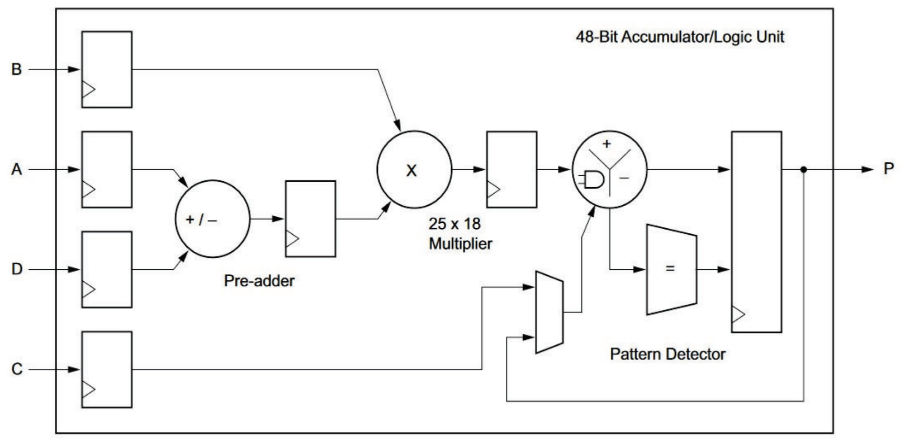


Figure 2.7: Xilinx series-7 family embedded DSP block diagram (Courtesy of Xilinx).

Storage elements

Advanced FPGAs include embedded memory elements such as block RAMs (BRAMs), LUTs, and shift registers (SRLs). Due to the flexibility of the LUT architecture in Xilinx FPGAs, these blocks can be used as 256-bit memories and are commonly referred to as distributed memories. This is the fastest kind of memory available on the FPGA device because it can be instantiated in any part of the fabric that improves the performance of the implemented circuit. A shift register consists of a chain of registers connected to each other. This structure is to provide data reuse along a computational path. Other than the distributed RAMs, the series 7 FPGAs of Xilinx, consist of embedded block memories [57]. The block RAM in Xilinx Virtex-7 family FPGAs stores up to 36Kbits of data and can be configured as either two independent 18Kbit RAMs or one 36Kbit RAM. Each 36Kbit block RAM can be configured as a 64Kbit RAM by cascading two adjacent block RAM.

Phase-Locked loops (PLL)

Embedded Phase-Locked loops in contemporary FPGAs like the Xilinx series 7 family are used for driving the FPGA fabric at a fine-tuned clock.

High-speed transceivers

The high-speed transceivers are used to implement high-speed data transfer protocols.

2.3.2 FPGA design principles

The key difference between a processor and an FPGA is that a processor is a fixed hardware architecture. A compiler translates a program into a set of machine-language codes understandable for the CPU. The CPU executes the machine code sequentially to perform the desired algorithm. FPGAs, on the contrary, are a blank slate of building blocks. A synthesis tool configures these building blocks to build specific hardware that performs the desired computations. Since FPGAs are reprogrammable, they are well suited used for rapid prototyping.

Parallelism

Parallel computation is possible at three levels.

- Algorithmic-level
- Hardware duplication
- Pipeline-level

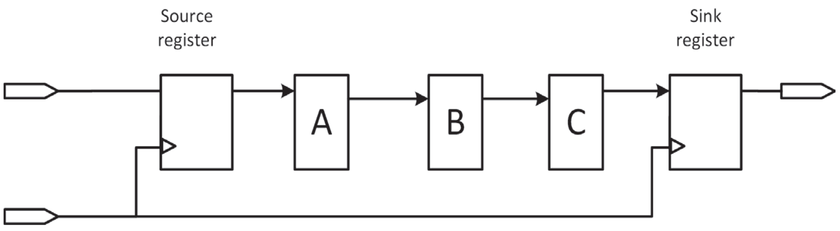
Parallelisation at the algorithmic level is basically a hard problem, as it is nontrivial to parallelise a sequentially described algorithm. A thorough investigation of the algorithm is necessary to find if it is feasible or not. As a simple rule of thumb, all functions that do not have data dependencies on the output of other functions may be easily parallelised.
Another approach to parallelisation is to implement the same hardware multiple times, which is simple and effective to improve the throughput. However, this approach is often expensive and a trade-off between the costs and performance must be considered.
The third approach, pipeline parallelism, uses a pipeline, which can process several data streams in parallel. This is a special case of the pipelining which is described in the following.

Latency

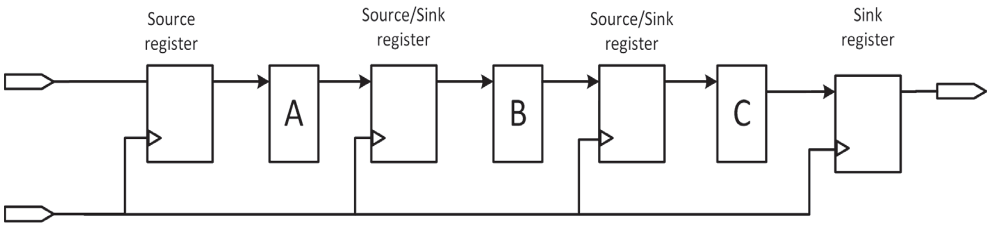
Latency is the number of clock cycles needed to complete an instruction or a set of instructions to generate a result value. That is, the result of the application is not available until the latency time is passed from the start time. The latency is a key performance metric in hardware design. In most cases, the problem of latency can be resolved through the use of pipelining.

Pipelining

The concept of pipelining is based on the observation that a new input can be supplied to the circuit not only after the circuit evaluation is completed for the previous input, but much earlier, if the new input does not depend on the output of the computation with the first input. In the RTL methodology, this concept is transformed to introduce additional registers into the circuit to shorten the worst-case combinational logic delay between two register stages. Then the second input may be supplied to the circuit as soon as the now shorter worst-case latency has elapsed. Figure 2.8 illustrates non-pipelined versus pipelined design. In the pipelined design, two extra registers have been placed between combinational logic A, B, and C. realising a two-stage pipeline design. Here, the long path delay from A to C (Source to Sink register) is shortened (divided by 3). As a result, a higher clock frequency can be applied to the system. The pipeline technique improves the clock frequency. However, it may have a drawback. Data dependencies may introduce pipeline stalls, when no new input can be supplied, for example, when the previous computation is not finished yet. The best case is achieved, when the clock frequency increases without introducing pipeline stalls, then the performance scales linearly with the clock frequency.
Throughput is another metric used to determine overall performance of hardware design. It is the number of clock cycles required for the processing logic to accept the next input data. For example, figure 2.8 shows two implementations. If the logic delay of blocks A, B, and C is equal to 5 ns (nanoseconds) each, the none pipelined implementation (upper one) requires 15 ns clock cycle between input data samples, while the pipelined design (the bottom one) needs only 5 ns between input data samples. It is clear that the second implementation has



(a) Non-pipelined design.



(b) Pipelined design.

Figure 2.8: Pipelined vs. none pipelined design.

a higher performance, because it can accept a higher input data rate. The clock period and throughput can be determined as functions of pipeline stages n.

$$\begin{aligned} T_{clk} &\propto \frac{1}{n} \\ Throughput &\propto n \end{aligned} \tag{2.49}$$

A critical path is defined as the path between an input and an output with the maximum delay.

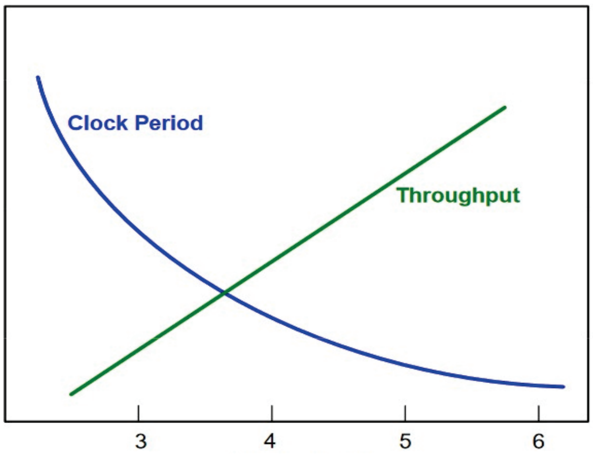


Figure 2.9: Circuit clock period and Throughput versus number of pipeline stages (n).

The pipeline technique must be used when a critical path delay does not meet the design timing requirements. Parallelism can be used when the critical path is bounded by the circuit timing constraints.

2.3.3 Algorithm-Specific Optimisations

As discussed, different types of optimisations can be generally applied to most of algorithms and implementations. However, it is often possible to optimise specific parts of an algorithm.

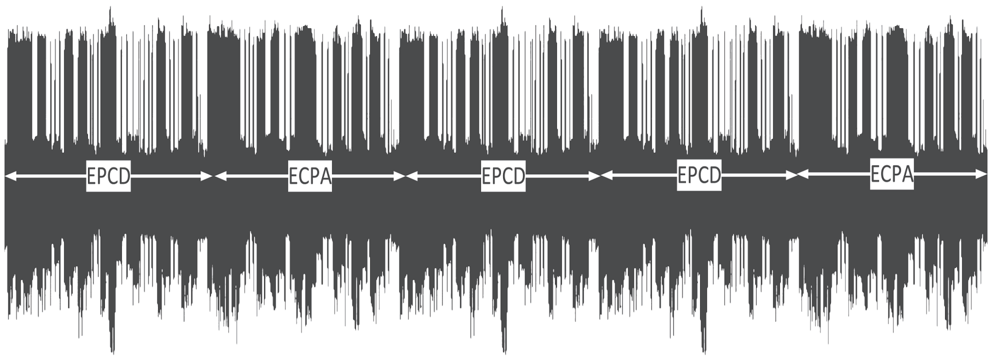


Figure 2.10: Sample ECC side-channel power signal.

For example, two mathematically equivalent functions can be implemented by smaller or faster circuits. Another approach is to instantiate FPGA primitives for a section of an algorithm. Such algorithm-specific optimisations often lead to notable improvements because only a small subset of an optimisation problem is considered, and the problem is approached with the specific optimisations. However, an optimal division into sub-problems is itself very difficult and thus, manual intervention with domain-specific knowledge may improve the results.

2.4 Side-channel attacks

A cryptographic primitive can be imagined as a function parameterised by a key [58]. The implementation of this function on software using a set of instructions or on the hardware as a state-machine produces a pattern of changes in the power consumption and electromagnetic radiation of the embedded device. Monitoring this pattern change, known as side-channel data, can be used to elicit the secret information used by the primitive. Cryptographic primitives can be considered either as mathematical objects or as concrete devices executing a program. These two points of view lead to different approaches to break a cryptosystem. The first point of view is that of classical cryptanalysis. The second one is that of side-channel cryptanalysis. Even though side-channel cryptanalysis is specific to the implementation of the primitive to be broken, it is a very serious attack. Side-channel attacks have been used to break cryptosystems based on the primitives that are considered mathematically secure. The two cryptanalysis approaches can also be used together to create more powerful attacks.

2.4.1 Simple side-channel analysis

There are basically two types of side-channel analysis or attacks (SCA). Simple Side-Channel Analysis (SSCA) and Differential Side-Channel Analysis (DSCA). SSCA was first introduced by Kocher in [59]. SSCA —including Simple Power Analysis (SPA) if the measured leakage is the power consumption, and Simple Electro-Magnetic Analysis (SEMA) in case of measuring electromagnetic radiation— acquires traces of side-channel activity of an embedded device performing a cryptographic operation or any computation involving sensitive data. Figure 2.10 shows power signal trace obtained from an ECC core. The ECPD and ECPA operations are simply detectable. The attacker can analyse the side-channel data to discover the performed operations. Such leakage of data can possibly lead to the recovery of the whole key with a single trace and a single execution. Based on Kerckhoffs' principle [60], this attack assumes that the attacker has full knowledge of the performed algorithms. A side-channel leakage can even be used to perform reverse engineering and guess how an operation is implemented.

2.4.2 Differential side-channel analysis

A scalar multiplication algorithm that is protected against simple side-channel analysis may still be vulnerable to differential analysis. In most cases, the base group element is imposed by the system and the secret key is an ephemeral parameter varying at each execution. In some cases, however, the secret key is fixed and the base element is the input to the system. In such scenarios, DSCA becomes a threat.

Kocher et al. introduced Differential Side-Channel Analysis (DSCA) in [61]. Similar to SSCA, Differential Power Analysis (DPA) is a type of DSCA when power traces are measured, and differential electromagnetic analysis (DEMA), is a DSCA when the electromagnetic radiation is used for analysis. DSCA exploits the slight consumption leakages of a device to find data dependencies in the power consumption or electromagnetic radiation traces and recovers secret key by analysing its influence on the known data. DSCA requires a large number of samples in which the same secret key was used to operate on different data. It usually has two phases— data collection and data analysis. In the data analysis phase, extensive signal processing and statistical analysis are applied to the raw samples to extract the secret key. DSCA can be automated by using little or no information about the target implementation because it locates correlated regions in a device's power consumption. Although the knowledge of plain-text is not required, the DPA can use known plain-text or known cipher-text to find the secret key. In the following, a step-by-step differential side-channel analysis is described.

1. The attacker observes multiple cryptographic operations with different sets of data and captures power/electromagnetic traces.

2. The attacker records cipher-texts. Knowledge of plain-texts is not required.

3. The side-channel data is partitioned into subsets according to a property of the state being processed.

4. Statistical methods are applied to find differences in the subsets and determine whether a key block guess is correct.

5. Attacking is done one intermediate state after another until the output value state. Let b is the target bit, C is the corresponding observed cipher-text and k_g is the guessed key. The n collected power traces of s samples is denoted by $T[1 : n][1 : s]$ which corresponds to ciphers $C[1 : n]$. We define a selection function f which gets inputs b, C, k_g and can be either $f(C, b, k_g) = 0$ or $f(C, b, k_g) = 1$. if the guess for k_g is correct then the average power trace for $f(C, b, k_g) = 1$ would be slightly higher at the point of correlation and the average trace for $f(C, b, k_g) = 0$ would be slightly lower. However, if the key guess k_g is not correct, then the selection function $f(C, b, k_g)$ would be equal to the correct value for bit b with the probability of 50% for each cipher-text, yielding the average traces that are approximately equal. Let $\Delta_f[j]$ be the differential trace, which is computed between the two average traces. For an incorrect key guess k_g, $\Delta_f[j]$ should approach zero, and for a correct one, should approach the target bit's power contribution at the correlated samples. Then, the correct value of k_g can be identified from the spikes in its differential trace $\Delta_f[j]$. These assumptions and equations are

presented in the Kocher's original paper [61]. Figure 2.11 depicts the DPA attack on AES [62].

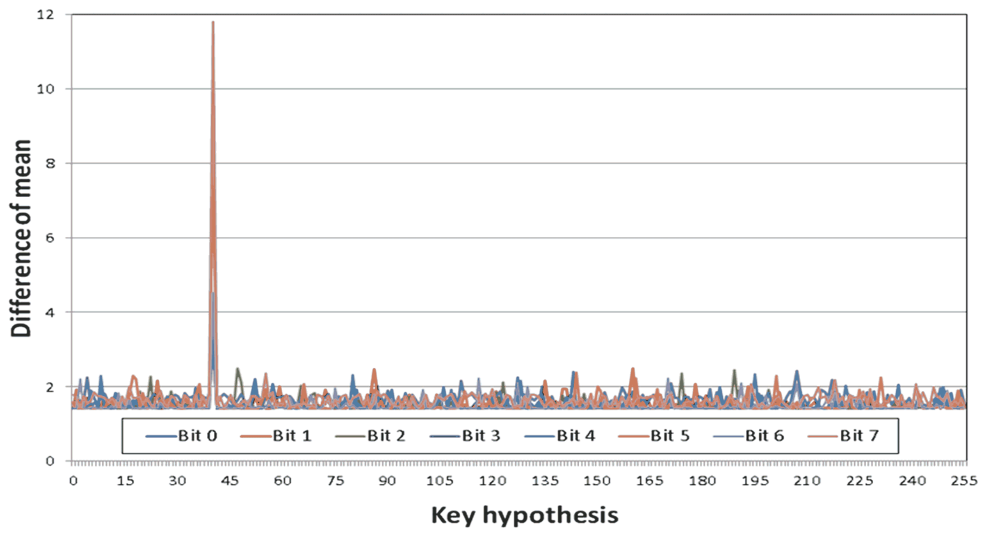


Figure 2.11: A sample of differential power analysis on AES hardware

Messeger et al. proposed three differential side-channel attack scenarios on public-key cryptosystems in [63], which are

- *Single Exponent Multiple Data Attack (SEMD)*

  SEMD assumes the attacker can exponentiate many random messages with at least one known (public) exponent and a secret exponent. The central idea of SEMD is the observation that by comparing the two obtained power signals (one taken from using public exponent, and the other one taken from using secret exponent), the attacker can see where they differ and thus learn the secret exponent [64]. Average signals are calculated and subtracted as in the mean method. This will make random data disappear, and only those signals dependant on the parameter will average out to two different values depending on the operation performed.

- *Multiple-Exponents, Single-Data Attack (MESD)*

  The assumption in MESD is that the attacker can choose the exponents. This approach improves Signal-to-Noise Ratio (SNR) and relies on the assumption that the attacker can exponentiate a constant value (which might not be known to him/her) repeatedly with the parameters chosen by him/herself.

- *Zero Exponent, Multiple Data Attack (ZEMD)*

  ZEMD assumes that the attacker knows the modulus and the algorithms use for exponentiation in the hardware but does not know the exponents.

Template DSCA

In the DSCA approach, the noise is considered as a hindrance that has to be reduced or eliminated. On the contrary, the template DSCA focuses on modelling the noise. Then, the noise model is utilised to extract the desired information from the side-channel data. Since this method utilises all available information in each sample for classification, it forms the strongest class of side-channel attacks [65]. The key requirement of the template attack is that the attacker must have access to the hardware to make desired changes. This process is called profiling. The attacker then uses profiling to capture precise and detailed noise signals which are called the template. This is much more powerful than an ordinary DSCA approach, where the noise is averaged.

2.4.3 Countermeasures against Side-Channel attacks

Most of the time, side-channel attacks are passive, which makes the detection of the attack almost impossible. A viable countermeasure for all side-channel attacks could be to prevent the side-channel data measurements. This requires applying aggressive shielding, which significantly increases the costs. Nevertheless, it cannot protect against invasive attacks. To date, many hardware and software countermeasures against side-channel attacks are introduced in the literature. The major difference between hardware and software countermeasures is that software countermeasures do not try to avoid data leakage by producing constant power consumption. Alternatively, they make side-channel information useless by adding random dummy operations. Bear in mind that a single countermeasure cannot cover a whole range of attacks. Commonly, it is required to combine several countermeasures to endure a specific range of attacks. Using a countermeasure is costly. Therefore, it is necessary to select countermeasures thoughtfully and compromise between security and performance [66].

Simple Side-channel attacks countermeasures

A good countermeasure is the design of cryptographic hardware with constant power consumption [67]. Such a countermeasure, however, is very hard to design and expensive to implement. In [68], Shamir proposed putting extra capacitors or batteries on the power supply path to alter the power consumption trace or make it constant in time. Practically, this approach is not a perfect solution. It reduces the size of the power bias. Accordingly, the number of traces required for a successful DPA attack increases yet remains vulnerable to DPA attacks.

A software countermeasure against DSA is the use of random interrupts. The CPU interleaves dummy instructions randomly with the cryptographic algorithm codes such that corresponding operation cycles do not match because of time shifts. This method smears the peaks across the differential trace due to the de-synchronisation effect, known as "*incoherent averaging*" in digital signal processing. Theoretically, using random interrupts do not make DSCA infeasible but considerably increases the number of samples required for a successful attack. Random dummy operations can be implemented at the hardware level as well. However, it is costly to implement true-randomisation in hardware. Countermeasures which are based on the injection of random noise to the power profile utilise redundant hardware. Benini et al. [69] used a method called *randomised power masking* that is the combination of power-management techniques and randomised clock gating to introduce a significant amount of non-deterministic noise to the power profile.

Inserting dummy arithmetics in group operations is a common solution to make a homogeneous operation flow. This countermeasure can be easily implemented in either software or hardware. It is generally an expensive countermeasure. A practical example of using dummy arithmetic is the Always-Double-and-Add ECC scalar multiplication, shown in Algorithm 14. This algorithm, named always-double-and-add ECC scalar multiplication, performs a dummy

Algorithm 14: Always Double-and-Add point multiplication algorithm

```
Input: $k, P$.
Output: $Q = k \cdot P$.
1  $m = \lceil \log_2 k \rceil$;
2  $Q = P$;
3  for $i = m - 2$ to $0$ do
4      $S = 2Q$;
5      $R = S + P$;
6      if ( $k_i = 1$) then
7          $Q = R$;
8      else
9          $Q = S$;
10 return $Q$;
```

ECPA operation after each ECPD when the scalar corresponding bit is '0' to make similar patterns of ECPD, ECPA for the scalar bit '1'. This countermeasure, however, does not protect against DCSA attacks performed by machine-learning algorithms [46]. Since the average zero bits for an k bit scalar is $\frac{k}{2}$, then this method adds extra latency of $\frac{k \times t_{PA}}{2}$ to computations, where t_{PA} is the latency of one ECPA operation. In [70], Chevallier et al. practised a different approach. They split ECPA and ECPD operations into several elementary basic blocks and then made them homogeneous. The advantage of this method is that operations are not unnecessarily made a lot longer. However, the measurement of the total the cryptographic operation time can still allow the attacker to obtain the total Hamming weight of the scalar. Moreover, the basic block operations are still distinguishable by the attacker.

Indistinguishable or unified addition and doubling formulae

This strategy applies to elliptic curve cryptography only. It is based on developing point-addition and point-doubling formulae, which are indistinguishable or unified [71]. This method may lead to utilising an alternative or equivalent form of an elliptic curve on which the point-addition and point-doubling formulae are unified. Hence, it can be applied to specific elliptic curves. The advantage of this countermeasure is that no dummy operations are involved.

Scalar multiplication with a fixed sequence of group operations

To thwart an SSCA on the elliptic curve cryptography hardware, point multiplication algorithms are used that have a fixed sequence of group operations and do not depend on the scalar value. Montgomery ladder point multiplication [41], shown in Algorithm 7, is an example of this approach. In this algorithm, both ECPD and ECPA operations are performed at each iteration. Unlike the Always-Double-and-Add algorithm, there is no dummy operation in the Montgomery ladder algorithm.

Scalar randomisation

The insertion of random decisions when the point multiplication algorithm is executing helps to prevent differential side-channel analysis. To be safe against SSCA, additional countermeasures can also be applied [72]. To be safe against SSCA, additional countermeasures can also be applied [72]. Three methods are used in the literature to randomise an ECC point multiplication [73].

- *Randomise scalar*.
  The order of an elliptic curve over the finite field $\mathbb{F}_p$ is the number of points on the curve denoted by $\#E(\mathbb{F}_p)$. A random integer r can be used to calculate the new scalar k' such that, $k' = k + r \times \#E(\mathbb{F}_p)$. For every point P on the elliptic curve E, since $\#E(\mathbb{F}_p)P = \mathcal{O}$, it concludes $Q = k \cdot P = k' \cdot P$. The drawback of this method is that depending on the bit-length of $r \times \#E(K)$, the effective key length may increase.

- *Randomise base point*.
  A point is blinded by adding a secret random point R for which one knows $S = k \cdot R$. Scalar multiplication is done by calculating $k \cdot (R + P)$ and subtracting S to get $Q = k \cdot P$. The points R and S can be stored in memory locations or registers and updated for each new execution as follows: $R = (-1)^b 2R$ and $S = (-1)^b 2S$, where b is a random bit. Note that there must be stored two additional points inside the device, which is also often not desirable.

- *Randomise using Projective coordinates*.
  Projective coordinates are used to avoid inversions in point multiplications. Point randomisation can be performed using projective coordinates as well because (X, Y, Z) and $(\lambda X, \lambda Y, \lambda Z)$ are representing same point $(x = \frac{X}{Z}, y = \frac{Y}{Z})$. A new λ can be chosen for every new execution.

Address-bit DSCA and countermeasures

Address-bit differential power or electromagnetic emission analysis exploits the variation in power or electromagnetic emission traces to distinguish the instructions used. It guesses individual bits of memory or registers' locations to detect operands reuse. Memory or registers' locations are a type of operands in CPU instructions. Therefore, access to different locations will provide different traces of side-channel data that are loosely correlated. On the contrary, the traces which correspond to the repeated access of the same memory location have higher correlations. This attack uses one bit of the memory address as a discriminating factor and splits all the memory addresses into two sets and then continues the same process. In [63], address-bit DPA and multi-bit DPA schemes are introduced. May et al. [74] suggested random register renaming on their designed processor dubbed NDISC that defeats address-bit DPA attack. Itoh et al. [75] designed an address-bit DPA attack on Montgomery elliptic curves. In [76], Itoh et al. proposed software implementation of a randomised-addressing countermeasure.

2.5 Applications of Machine-learning and Deep-learning in side-channel analysis

Recently, the application of machine learning in analysing side-channel data has been at the centre of attention [5]. Machine learning (ML) algorithms are powerful tools to detect change patterns in side-channel data and categorise operations performed in the embedded device. In a typical classification problem, the algorithm trains itself with a set of training data which consists of input data vectors (*features*) and their associated outputs (*labels*). This technique is called *supervised learning*. During the training process, the supervised learning algorithm makes assumptions based on the features and corrects itself if those assumptions do not match the corresponding labels. Eventually, a model is created that can classify the unseen input data accurately. In other terms, the generated model produces the correct result for the inputs other than training data. In contrast, in an unsupervised learning technique, no labels are available. The algorithm endeavours to extract attributes of the input data that are useful to relate the input data to a class of outputs.

Hospodar et al. [77] applied ML for side-channel analysis for the first time in 2011 targeting unprotected software implementation of AES. There are a couple of recent publications that studied advanced Deep Learning (DL) techniques as well. In supervised machine learning, the training dataset consists of N inputs $\vec{X}$, $x \in \mathbb{R}^n$, which is a vector of n features, and N' labels y_i, $i \in [1, N']$, $N' \leq N$. It tries to find a map from the input vector to a correct label y_j. Unsupervised learning algorithms, however, try to derive patterns and attributes of the input vector that best describe the output. The aim of any machine learning algorithm either supervised or unsupervised is to find a model $f(\vec{X})$ that defines a mapping from features to the correct outputs. It is achieved by adjusting the algorithm parameters to minimise a cost or error function E, which allows deciding the accuracy of the model applied to the input data. For side-channel analysis, a training input $\vec{X}$ is a vector of the power consumption or electromagnetic radiation samples over a specific time interval while a cryptographic function is performing by the target hardware. In the ML process, this raw data may be pre-processed before applying the ML algorithm. Most of the ML algorithms require normalised (re-scaled to values between ‘0’ and ‘1’) or standardised (having zero mean and unit variance) input data. The data points with the highest information content are then extracted or generated by combining or creating additional data. This process is called feature engineering. A proper learning algorithm will be selected, and its hyper-parameters, which control the algorithm’s behaviour, will be set. Finally, the performance of the selected model is verified by applying the test data. Test data is a part of data that is not used for training. Usually, a train/test-split assigns 80% of the sample data to the training set and 20% to the test set. However, if the test set is not sufficient, statistical uncertainty around the average test error can make the comparability of different ML algorithms difficult [78]. To avoid such a problem, the whole data set is split randomly into k disjunct subsets and one of them is used as the test dataset iteratively, and the rest is used as the training dataset. This method is called k-fold cross-validation. Cross-validation is also used for determining suitable hyper-parameters.

An ML model may experience *overfitting* or *underfitting* [78]. Over-fitting is when the model performs very well on the training set, but not on the test set. Underfitting happens when the model performs poorly on training. It means the model cannot predict the labels correctly. Modifying the model’s capacity is a way to control the degree of overfitting or underfitting. Capacity is the ability to fit a wide range of functions by changing the number or value of parameters

Dimensionality reduction

Dimensionality reduction is a feature engineering technique that reduces the number of features in the input dataset without having to lose much information and keep or even improve the model’s performance. This technique is used in both unsupervised and supervised learning machine learning algorithms to generate a lower-dimensional dataset as the input. However, the loss in the original dataset can adversely impact the model performance if the dropped information is crucial for the prediction. The most common method of dimensionality reduction is *Principal Component Analysis* (PCA). PCA projects data to the direction of increasing variance. The input data vector $\vec{X} = [x_1, \dots, x_n]$ is mapped to a new vector $\vec{Z} = [z_1, \dots, z_m]$, where $m < n$, by calculating the mean vector $\vec{\mu}$ on all input features of the original dataset.

The eigenvectors corresponding to the m largest eigenvalues of the $n \times n$-dimensional covariance matrix are called principal components. A new $n \times n$ matrix **A** is formed from the principal components. The transformed data is derived by calculating $\vec{Z} = \mathbf{A}^T(\vec{X} - \vec{\mu})$, and dropping the points which are corresponding to the $n - m$ smallest eigenvalues. PCA can also help to decrease the noise level in the side-channel data traces. However, side-channel data commonly deal with a high number of features that makes PCA computationally expensive [79]. Alternatively, *Points of Interest* (PoI) are widely used in the side-channel data analysis. These are the data points which correspond to the most leakage of information. POIs can be detected where the data in samples have the largest covariance.

2.5.1 Supervised Machine-learning algorithms

Supervised ML algorithms work either in classification or regression mode. In a classification model, the labels are in the form of finite classes. The algorithm is expected to specify which class is the best fit for the set of input features. Regression models, in contrast, carry out the prediction of quantitative outputs. In side-channel data analysis, ML is used to classify power signals to the set of operation classes. In the following, we describe some classification ML algorithms that can be used for SCA applications.

Decision trees

Decision Trees (DT) are supervised learning algorithms that classify the training data by creating a tree with zero or more internal nodes and at least one leaf node. Decision Trees are also the fundamental component of the Random Forests [80]. A decision tree is a flowchart-like composition in which any internal node represents a test on an attribute, each branch represents the outcome of this test, and each leaf node represents the decision taken after computing all attributes or the class label. The paths from the root to the leaf represent classification rules. Figure 2.12 illustrates a simple DT that performs an AND operation on a three-input example $\vec{X} = [x_1, x_2, x_3]$.

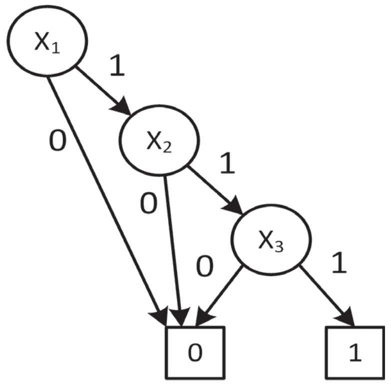


Figure 2.12: A Decision Tree performing 3 bit AND operation

Random forest

Random Forest (RF) consists of a large number of individual decision trees that operate as an ensemble [81]. Each tree is constructed only with a random subset of available features (typically the square root of the total number of features or even only one feature [82]). As shown in figure 2.13, the prediction is based on the majority of votes from each of the decision trees. The class which gets more votes becomes the winner of the model's prediction. Hence, the procedure is relatively robust against outliers and noise and can easily be parallelised. Random forest is used to identify the most important features of the training dataset. The larger number of decision trees leads to a more accurate result. The advantage of the Random forest is that it avoids overfitting.

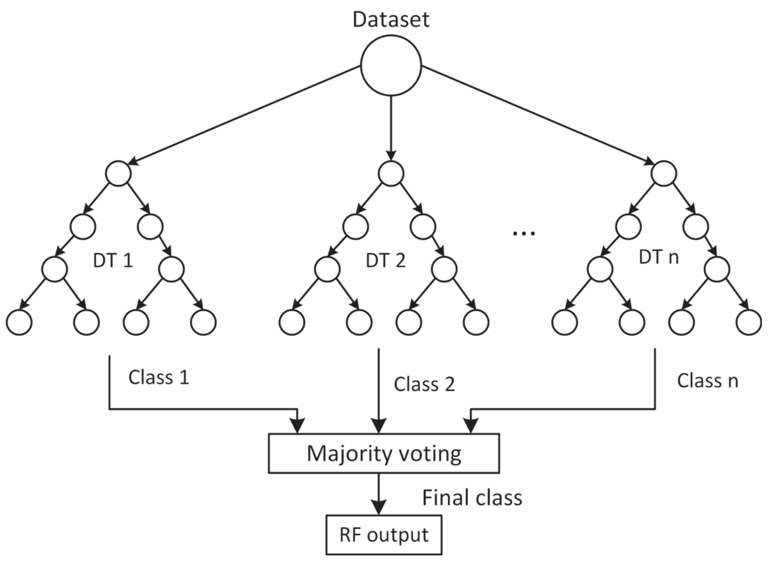


Figure 2.13: Random Forest ensemble classification

K-Nearest Neighbours

K-nearest neighbours(kNN) is a simple non-parametric algorithm. It stores all available cases and classifies new cases based on similarity measures such as distance functions. It does not make any assumptions on the underlying data distribution. The non-parametric property of kNN can be helpful in classification problems; since most of the times, data does not follow the theoretical hypotheses. Therefore, kNN could be one of the first choices for a classification study when there is little or no prior knowledge about the distribution of the data. In the kNN algorithm, an object is classified by the majority vote of its neighbours with the object being assigned to the class most common among its k nearest neighbours (k is a positive integer, typically small). If k = 1, then the object is assigned to the class of its nearest neighbour. Figure 2.14 illustrates how kNN algorithm works [83].

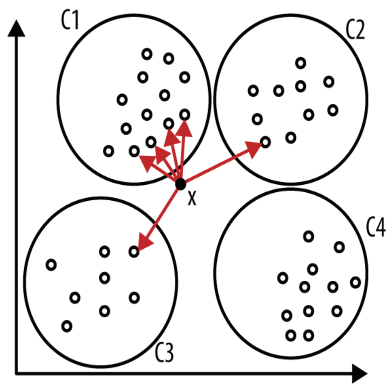
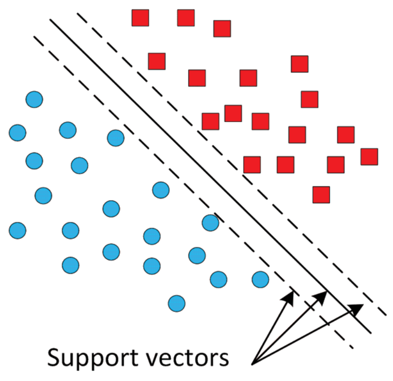


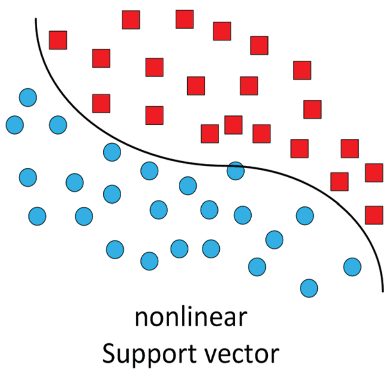
Figure 2.14: kNN classification with four classes, $\{C1, C2, C3, C4\}$

Support Vector Machine

Support Vector Machine (SVM) is one of the most popular supervised learning algorithms. The support vector machine algorithm aims to find a hyperplane in an N-dimensional space (N is the number of features) that distinctly classifies the data points. Many possible hyperplanes could be chosen to separate the two classes of data points. The objective is to find a plane that has the maximum margin, that is the maximum distance between the data points of both classes. This is illustrated in Figure 2.15a. Maximising the margin distance provides some reinforcement so that future data points can be classified with more confidence. In the SVM classifier, it is easy to have a linear hyperplane between these two classes. However, another question which arises is whether we should add this feature manually to have a hyperplane. The answer is negative, SVM algorithms have a technique called the kernel trick. The SVM kernel is a function that takes low dimensional input space and transforms it into a higher-dimensional space, i.e. it converts a not separable problem into a separable problem. It is mostly useful in non-linear separation problems. Figure 2.15b shows a nonlinear hyper-plane used to separate to class of data.



(a) Linear SVM



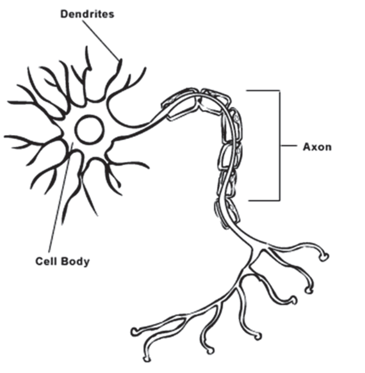
(b) Non-linear SVM

Figure 2.15: Linear and non-linear Support Vector Machine

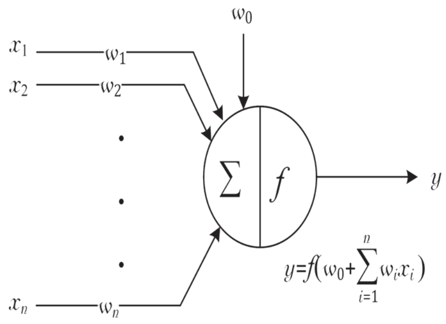
2.5.2 Artificial neural networks and deep learning

Artificial neural networks (ANN) are modelled on the parallel architecture of animal brains, not necessarily human ones [84]. A *neuron* is a cell that can transmit and process electrochemical signals. Each neuron is connected with other neurons to create a network. Within the human body, there are a large number of neurons interconnected with each other. Figure 2.16a shows the structure of a neuron which consists of an input (known as the dendrite), a cell body, and an output (called the axon). The outputs of the neurons are connected to inputs of other neurons and develop a network. A neuron is activated when an electrochemical signal is received by a dendrite. The cell body resolves the weight of the signal; if a threshold is passed, the signal will continue to pass through to the axon. An axon functions as an electric cable, transmitting the signal to the next stage.

The basis of ANN is the *perceptron*. A Perceptron operates on numbers. When a number or a vector of numbers is passed to the input, it feeds a function that calculates the outgoing value. This function named the *activation function*. A perceptron node can accept any number of inputs [84]. As illustrated in Figure 2.16b, it receives a vector of input features $\vec{X} = [x_1, \dots, x_n]$ and performs a linear combination using the weight values $w_1, \dots, w_n$ of its input connections and a bias value w_0. These weights are adjusted according to the training dataset when the perceptron is in learning mode. The result is then passed to a threshold activation function f (for example, the Hyperbolic tangent $f(x) = \tanh(x)$)) to calculate the output value $\tilde{y}$. The most common activation functions are Sigmoid, Hyperbolic tangent, and ReLU (Rectifier Linear Unit) [84].



(a) A human neuron



(b) A perceptron

Figure 2.16: Linear and non-linear Support Vector Machine

Single-layer perceptrons are only able to represent functions whose underlying data set is linear separable such as Boolean functions [5]. Multiple layers of perceptrons are stacked together and form a network called *multi-layer perceptrons* (MLPs). MLPs extend the application of neural networks to more complex problems. An MLP consists of three distinct types of layers, as shown in Figure 2.17. The input layer receives input features (raw data). The number of perceptrons in an input layer corresponds to the number of input features. All outputs of the input layer are connected to each perceptron of the following hidden layer. The number of hidden layers and perceptrons per each depends on the model design. Usually, too many hidden layers can lead to overfitting and underestimating the number of hidden layers may result in underfitting. The number of perceptrons in the output layer corresponds to the classes of the problem to solve.

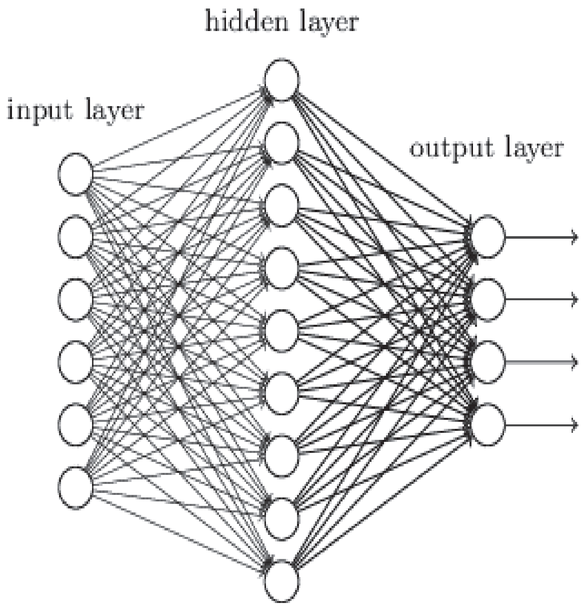


Figure 2.17: Multi-layer perceptrons including input, output, and hidden layers.

Deep-learning

Deep learning (DL) is a subset of machine learning methods based on artificial neural networks. Deep learning architectures are mainly categorised in three classes.

The fully connected networks are the basic type of neural networks. The major advantage of fully connected networks is that they are “structure-agnostic”. That is, no special assumptions need to be made about the input data. A fully connected network can be described as a function $f : \mathbb{R}^n \to \mathbb{R}^m$ such that any of m outputs depends on the n inputs. The simplest form of a fully connected neural network is the perceptron [85].

The features extractor networks are often used in image recognition and classification. The goal of these networks is to learn higher-levels and deep features of data that are most useful for classification or target detection. This can be done via computing a convolution between the data and some filters followed by a down-sampling operation to keep only the most informative features. A typical example of features extractor networks is the *Convolutional Neural Network* (CNN) [86]. In recent studies, one-dimensional CNNs (1D-CNN) have demonstrated superior performance on the applications which have limited labelled data and high signal variations acquired from different sources [87]. Some examples of these applications are the interpretation of biological signals like ECG (Electro-Cardio-Gram), civil, mechanical or aerospace structures monitoring, and high-power circuitry, power engines or motors monitoring and fault finding). The 1D-CNN is a powerful algorithm for deriving features from a fixed-length segment of the overall dataset, where the location of the feature is in the segment is not important. Typically, CNN is composed of alternating layers of

1. locally connected convolutional filters and
2. down-sampling, (known as *pooling*), followed by
3. a fully connected layer that works as a classifier (usually a *SoftMax* layer).

The time dependency networks are a set of neural networks that differ from the other ones in their ability to process information shared over several time-steps. In a traditional neural network, it is assumed that all inputs and outputs are mutually independent. However, for various applications, this assumption is not practical for many applications. The core idea of time dependency networks is that each neuron will infer its output from both the current input and the output of previous neurons. This feature is quite interesting in side-channel data analysis since the leakage is spread over several time samples. The *Recurrent Neural Networks* (RNN) [88] and especially the *Long-Short-Term-Memory* units (LSTM) [89] are samples of time dependency neural networks.

2.5.3 Related works in the application of deep learning for side-channel attacks

The recent works on side-channel data analysis using DL techniques are categorised as follows.

1. *Defeating SCA protected and unprotected symmetric cryptographic implementations.* In [90], Maghrebi et al. demonstrated that the Deep-Learning based SCA (DL-SCA) is very efficient to break both unprotected and masked AES implementations. The authors experienced several types of DL models (MLP, CNN, LSTM, and stacked Auto-Encoders [91]). Their results proved the advantage of the Deep-learning technique on the well-known template attacks. Later, Cagli et al. proposed an end-to-end profiling approach based on CNN that is efficient in the presence of trace misalignment [92]. This property helps to streamline the evaluation process as no pre-processing of the traces is needed. Prouff et al. [93] revisited different methodologies to select the most suitable hyper-parameters, which are the parameters that define a DL configuration. For example, the number of layers, the number of epochs, etc. More interestingly, the authors published an open database, named ASCAD, that contains electromagnetic traces of a masked AES implementation along with the source code of the used neural network architectures.

2. Defeating secure asymmetric cryptographic implementations.
   In [94], the authors presented profiling SCA against a secure implementation of the RSA algorithm. The target implementation was protected with classical side-channel countermeasures (blinding of the message, blinding of the exponent and blinding of the modulus). Through their practical experiments, the authors pointed out the high potential of deep learning attacks – in particular the CNN models–against secure RSA implementations.

3. *Using the DL-SCA in a non-profiling context.* Timon utilised Prouff's [93] ASCAD database to devise first non-profiled side-channel analysis using deep-learning algorithms. His idea was to produce hypothetical intermediate values over all key hypotheses. For each guessed key using the calculated intermediates as labels, a deep-learning training was performed. If the key guess is correct, then training metrics (i.e. accuracy and loss) are significantly better than the trials where the key guess is wrong. Furthermore, he demonstrated that data augmentation proposed in [92], can be used in combination with MLPs to improve the performance of the analysis.

4. *Using DL as a Point of Interest (PoI) selection method*. In several works, researchers used DL as a leakage assessment method [79, 95, 96]. Masure et. al. [96] suggested that the gradient of the loss function can be analysed during the training. The application

of the well-known attribution methods used in [79] and the exploitation of the sensitivity analysis techniques in [95]. The results have shown that Deep-learning methods based on PoI selection and state-of-the-art leakage assessment methods have similar performance.

3 RNS Arithmetic Hardware

3.1 Introduction

In this chapter, we first discuss an optimal RNS moduli selection with an aim to accelerate the hardware. Then, we introduce improved hardware implementations for RNS addition, subtraction, and multiplication. In the following, we propose an improvement of the RNS Montgomery modular reduction algorithm and introduce a new RNS modular reduction algorithm based on the sum of residues. Finally, we provide hardware implementation variants and performance parameters.

3.2 Choosing RNS bases for cryptography applications

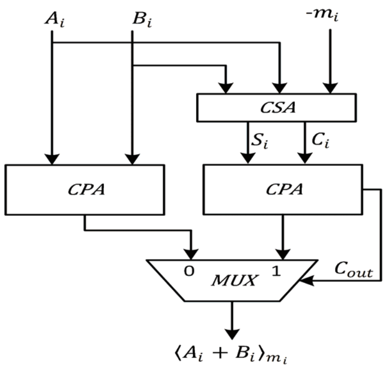
Modern public-key cryptography algorithms rely on large integer arithmetic. It is essential to use an RNS with at least double the bit-length dynamic range of the finite field characteristic. In this research, we work on 256-bit prime fields. Thus, the dynamic range of the RNS base must be at least 512 bits. In practice, we need a larger dynamic range due to

1. Calculations are in range $[-p, p]$, that is mapped to the range $[0, 2p]$ in RNS.
2. In the Point doubling equations (2.31) the term $(8{Y_1}^4)$ is at most 515 bits before performing a reduction.
3. As discussed in Subsection 2.1.2, the effective dynamic range is reduced by the boundary condition (2.17).

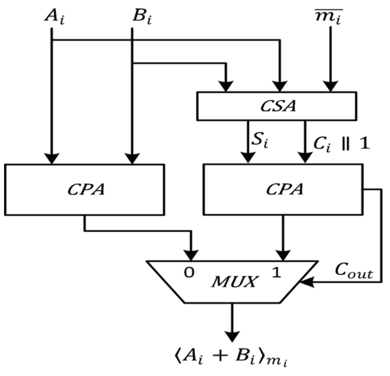
Later, in this chapter, we will show that the minimum required dynamic range to be on the safe side is 520 bits. There are two approaches to choose the moduli set suitable for cryptographic applications. The first solution is to choose more RNS channels with smaller bit size co-primes. As an example, in [97], forty RNS channels of 14-bit co-primes are used to build a 560-bit dynamic range. The first impression is that the hardware implementation is smaller and more efficient in this case. However, it is not easy to find forty optimal same bit-size co-primes, and it

Table 3.1: 66-bit co-primes of moduli set $\mathcal{B}$.

$2^{66}-1$	$2^{66}-2^2-1$	$2^{66}-2^3-1$	$2^{66}-2^4-1$
$2^{66}-2^5-1$	$2^{66}-2^6-1$	$2^{66}-2^8-1$	$2^{66}-2^9-1$



(a) Modular adder architecture



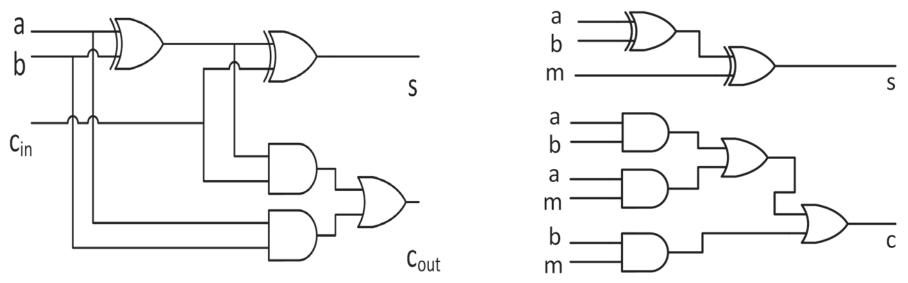
(b) Modified modular adder architecture

Figure 3.1: RNS adder design adder architectures

is inevitable to use non-optimal RNS bases. Further, co-primes with different bit-sizes lead to inefficient hardware implementations. As the experience in [97], [98] showed, huge hardware resources are required to implement this configuration. The second approach, that is most common in the literature, is using fewer RNS channels with larger bit-size co-primes. For 256-bit prime field arithmetic, eight channels of bit-length greater than 64 is the most common RNS selection in the literature. Bajard et al. in [99] studied the properties of optimal RNS bases in general form $m_i = 2^n - 2^{t_i} \pm 1$, $t_i < \frac{n}{2}$ for applications in cryptography. In this research, we used a moduli set $\mathcal{B}$ with eight, 66-bit pseudo-Mersenne co-primes in the form of $2^{66} - 2^{t_i} - 1$, as shown in Table 3.1. This moduli set provides a 528-bit dynamic range that is suitable for our 256-bit prime field ECC arithmetic. In the following, we introduce efficient RNS hardware arithmetic units used in this thesis to build the RNS ECC co-processor.

3.3 RNS Addition

This architecture is fast, since the two CPA adders operate in parallel. However, it costs one carry-save adder (CSA) and two carry-propagate adders (CPA). Figure 3.2 depicts one-bit full adder and one-bit carry-save adder at the gate level. Assuming that all gates have the same area, implementation an n bit RNS adder requires $17n$ gates and one n-bit 2:1 multiplexer. Using the optimal moduli of Table 3.1, it is possible to reduce the logic efficiently and improve the speed at the same time. All the bits of an integer in the form $m_i = 2^n - 2^{t_i} - 1$ is one except the bit t_i. The inverse of m_i denoted by $\overline{m_i}$ is 2^{t_i}. Instead of calculating or saving 2's complement of m_i, it is preferred to use its inverse and set the carry-in of the CPA adder to one. This is shown in Figure 3.1b. Since the XOR of any logic to '1' is the inverse of the logic and the XOR of a logic to '0' is the logic itself, the CSA adder logic is using Boolean reduction



(a) Full adder architecture (b) CSA adder architecture

Figure 3.2: Architecture of CPA and CSA adders at gate level.

equations in (3.1)

$$\begin{aligned}
a \oplus b \oplus \text{`0'} &= a \oplus b \\
(a \wedge b) \vee (a \wedge \text{`0'}) \vee (b \wedge \text{`0'}) &= a \wedge b \\
a \oplus b \oplus \text{`1'} &= a \odot b \\
(a \wedge b) \vee (a \wedge \text{`1'}) \vee (b \wedge \text{`1'}) &= a \vee b,
\end{aligned} \tag{3.1}$$

the output of the CSA adder for channel i is reduced to

$$\begin{aligned}
S_i &= (A_i^{n-1} \oplus B_i^{n-1})||(A_i^{n-2} \oplus B_i^{n-2}) \cdots ||(A_i^{t_i} \odot B_i^{t_i}) \cdots ||(A_i^{0} \oplus B_i^{0}), \\
C_i &= (A_i^{n-1} \wedge B_i^{n-1})||(A_i^{n-2} \wedge B_i^{n-2}) \cdots ||(A_i^{t_i} \vee B_i^{t_i}) \cdots ||(A_i^{0} \wedge B_i^{0}).
\end{aligned} \tag{3.2}$$

This implementation requires only $2n$ gates per channel for CSA. Thus, the RNS adder implemented using $12n$ gates per channel. The inputs of the right CPA adder in Figure 3.1b are S_i and C_i. The bit C_i^0 is set to one.

3.4 RNS Subtraction

RNS subtraction $\langle A_i - B_i \rangle_{m_i}$ is basically addition of A_i to the additive inverse of B_i that is, $(m_i - B_i)$. The 2'complement of B_i, is the increment of its inverse. Thus, $-B_i$ can be replaced with $\overline{B_i} + 1$. The increment one is implemented by setting adder's carry-in to '1'. Our proposed hardware for the implementation of RNS subtraction is shown in Figure 3.3a. Similar to the RNS addition circuit, the input carry bits of CSA adders are set to 1. The values of $A_i + \overline{B_i}$ and $A_i + \overline{B_i} + m_i$ are calculated in parallel and the correct result is chosen by a 2:1 multiplexer. In our selected moduli set $\mathcal{B}$, all the bits of m_i are ones, except the t_i-th bit. Therefore,

$$\begin{aligned}
SR_i &= (A_i^{n-1} \oplus B_i^{n-1})||(A_i^{n-2} \oplus B_i^{n-2}) \cdots ||(A_i^{t_i} \odot B_i^{t_i}) \cdots ||(A_i^{0} \oplus B_i^{0}), \\
SL_i &= (A_i^{n-1} \odot B_i^{n-1})||(A_i^{n-2} \odot B_i^{n-2}) \cdots ||(A_i^{t_i} \odot B_i^{t_i}) \cdots ||(A_i^{0} \odot B_i^{0}).
\end{aligned} \tag{3.3}$$

Similarly,

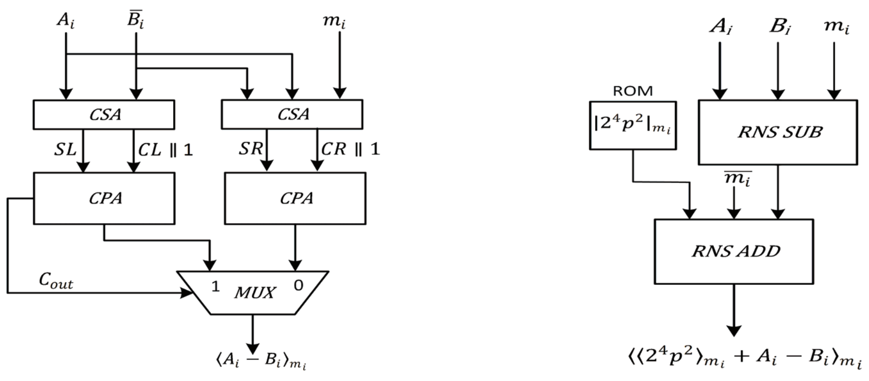
$$\begin{aligned}
CR_i &= (A_i^{n-1} \vee \overline{B_i}^{n-1})||(A_i^{n-2} \vee \overline{B_i}^{n-2}) \cdots ||(A_i^{t_i} \wedge \overline{B_i}^{t_i}) \cdots ||(A_i^{0} \vee \overline{B_i}^{0}), \\
CL_i &= (A_i^{n-1} \wedge \overline{B_i}^{n-1})||(A_i^{n-2} \wedge \overline{B_i}^{n-2}) \cdots ||(A_i^{t_i} \wedge \overline{B_i}^{t_i}) \cdots ||(A_i^{0} \wedge \overline{B_i}^{0}).
\end{aligned} \tag{3.4}$$

Comparing (3.2) and (3.3), it is observed that SR_i is same as S_i and is the bitwise inverse of SL_i except for the t_i-th bit, that is same in all S_i, SR_i, SL_i. The hardware implementation of RNS add/subtract is more efficient using this shared logic.

The RNS is not a positional numbering system. Sign detection is a costly operation in RNS. Therefore, the sign detection in RNS subtraction should be avoided. In the following, we further explain our method to avoid sign detection. Consider the subtraction $S = A - B$. As A and B are representing numbers not greater than 515 bits in our RNS ECC core design, we add a factor of p to A to ensure that the result of the subtraction is always a positive integer. The coefficient κ must be chosen such that the result $\kappa \cdot p$ is greater than 515 bits. Here, we chose $\kappa = 2^4 \cdot p$ which results in the 516-bit integer $\kappa \cdot p$. It is required to perform a modular reduction after each subtraction to get the correct result. That is,

$$S = (2^4 \cdot p^2 + A - B) \mod p = (A - B) \mod p. \tag{3.5}$$

Figure 3.3b illustrates the implementation of an RNS modular subtraction unit.



(a) RNS subtraction block diagram (b) Modulus-p RNS subtraction block diagram

Figure 3.3: Subtraction in RNS

The implementation results of RNS adder and field p subtractor on different Xilinx FPGAs are shown in Table 3.2.

Table 3.2: Area and Delay of RNS channel addition/subtraction on different FPGAs

Unit	**Device**	**Max. Logic delay** (ns)	**AREA** LUTs / DSP / FF
RNS Add	ARTIX 7	6.017	196 / 0 / 0
RNS field p subtract	ARTIX 7	7.61	331 / 0 / 0
RNS Add	VIRTEX 7	4.931	193 / 0 / 0
RNS field p subtract	VIRTEX 7	4.225	330 / 0 / 0
RNS Add	KINTEX 7	4.202	199 / 0 / 0
RNS field p Subtract	KINTEX 7	4.393	331 / 0 / 0
RNS Add	VIRTEX UltraScale+	2.139	199 / 0 / 0
RNS field p Subtract	VIRTEX UltraScale+	3.910	330 / 0 / 0
RNS Add	KINTEX UltraScale+	2.018	204 / 0 / 0
RNS field p Subtract	KINTEX UltraScale+	3.842	330 / 0 / 0

3.5 RNS Multiplication

When computing modular additions and subtractions in Section 3.3 and 3.4, we assumed that both operands A and B are n-bit integers within the range $[0, m-1]$. The result can be kept within the range by simply adding or subtracting the modulus m. Modular multiplication of two n-bit numbers, however, is the remainder of the product $X = A \cdot B$ divided by the modulus m. This operation is also called the reduction of X modulo m.

3.5.1 Modular reduction

Barrett proposed a modular multiplication algorithm [11] in 1984. A modification on the Barrett method was suggested by Cao [100] in 2014 which utilised base 2 to make the algorithm more suitable for the hardware implementation. Suppose X is a $2n$-bit integer to calculate $r = X \bmod m$, where m is an n-bit modulus. The quotient q can be written as

$$q = \left\lfloor \frac{X}{m} \right\rfloor = \left\lfloor \frac{\frac{X}{2^n} \times \frac{2^{2n}}{m}}{2^n} \right\rfloor. \tag{3.6}$$

The estimate of the quotient q, $\hat{q}$ is defined as

$$\hat{q} = \left\lfloor \frac{\left\lfloor \frac{X}{2^n} \right\rfloor \times \left\lfloor \frac{2^{2n}}{m} \right\rfloor}{2^n} \right\rfloor. \tag{3.7}$$

The term $\mu = \left\lfloor \frac{2^{2n}}{m} \right\rfloor$ is constant and can be pre-computed and saved in a look up table. We define $\epsilon = \frac{X}{2^n} - \left\lfloor \frac{X}{2^n} \right\rfloor$ and $\tau = \frac{2^{2n}}{m} - \left\lfloor \frac{2^{2n}}{m} \right\rfloor$, $0 \le \tau, \epsilon < 1$. From (3.7) it can be observed that

$$\hat{q} \le q = \left\lfloor \frac{\left(\left\lfloor \frac{X}{2^n} \right\rfloor + \epsilon\right) \times \left(\left\lfloor \frac{2^{2n}}{m} \right\rfloor + \tau\right)}{2^n} \right\rfloor. \tag{3.8}$$

From the definition of ϵ and τ, it is concluded that

$$\hat{q} \le q \le \left\lfloor \frac{\left\lfloor \frac{X}{2^n} \right\rfloor \times \left\lfloor \frac{2^{2n}}{m} \right\rfloor}{2^n} + \frac{\left\lfloor \frac{X}{2^n} \right\rfloor + \left\lfloor \frac{2^{2n}}{m} \right\rfloor + 1}{2^n} \right\rfloor. \tag{3.9}$$

The term on the left-hand side of the plus sign in (3.9) is $\hat{q}$ and the maximum value of the term on the right-hand side is 3. In conclusion, the value of q falls between $\hat{q}$ and $\hat{q} + 3$. As both q and $\hat{q}$ are integers, then

$$0 \le \hat{q} \le q \le \hat{q} + 2. \tag{3.10}$$

The remainders r and $\hat{r}$ are

$$\begin{aligned} r &= X - qm \\ \hat{r} &= X - \hat{q}m \end{aligned} \tag{3.11}$$

From (3.10) and (3.11) it is deduced that $r = \hat{r} - \lambda m$, $\lambda \in \{0, 1, 2\}$.

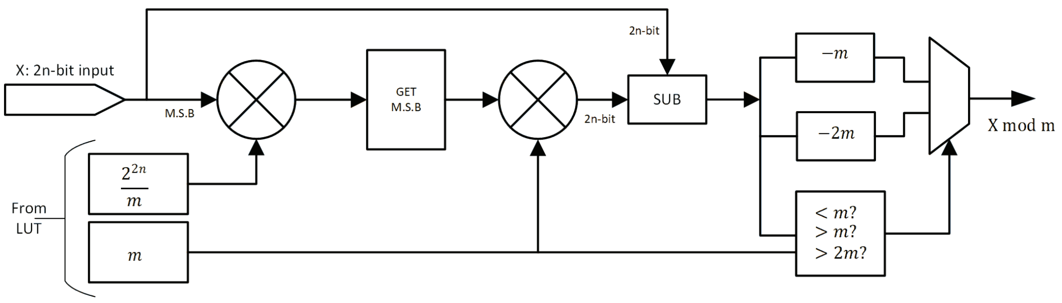


Figure 3.4: Implementation of a Barrett reduction $X \mod m$

Consequently, hardware implementation of Barrett modular reduction consists of two $n \times n$ multipliers, a comparator, and n-bit memory to save constant μ. Figure 3.4 shows the block diagram of the Barrett modular reduction hardware implementation. The long data-path from the input to the output limits the speed of the hardware. The modular reduction is the most critical operation in cryptographic hardware design. The performance of the reduction unit directly impacts the overall performance of the system. Barrett reduction is the only choice for the modular reduction for a non-optimal modulus.

Implementation of the modular reduction using optimal moduli in general form $m_i = 2^n - 2^{t_i} - 1$, $t_i < \frac{n}{2}$ is very fast and low-cost on hardware [101]. Suppose X is a $2n$-bit integer $(0 \le X < 2^{2n})$. It can be broken up into two n-bit most significant and least significant integers denoted as X_H and X_L, respectively. In other terms, $X = X_H 2^n + X_L$.

Since $\langle 2^n \rangle_{(2^n - 2^{t_i} - 1)} = 2^{t_i} + 1$, then

$$\langle X_H 2^n + X_L \rangle_{(2^n - 2^{t_i} - 1)} = \langle \langle X_H 2^{t_i} \rangle_{(2^n - 2^{t_i} - 1)} + X_H + X_L \rangle_{(2^n - 2^{t_i} - 1)}. \tag{3.12}$$

The term $X_H 2^{t_i}$ has $(n + t_i)$ bits and can be re-written as $X_H 2^{t_i} = X_{HH_i} 2^n + X_{HL_i}$.
Let $(x_{n-1} \dots x_0)$, $x_i \in \{0, 1\}$ be the binary representation of X_H. Then we introduce X_{HH_i} as the most significant t_i bits of X_H, i.e. $(x_{n-1} \dots x_{n-t_i-2})$ and X_{HL_i} as the rest least significant bits $(x_{n-t_i-1} \dots x_0)$ left shifted t_i times, i.e. $X_{HL_i} = (x_{n-t_i-1} \dots x_0 \underbrace{0 \cdots 0}_{t_i\ zeros})$.

Similarly,

$$\langle X_{HH_i} 2^n \rangle_{(2^n - 2^{t_i} - 1)} = \langle X_{HH_i} 2^{t_i} + X_{HH_i} \rangle_{(2^n - 2^{t_i} - 1)}. \tag{3.13}$$

Since X_{HH_i} is t_i bits long, the term $X_{HH_i} 2^{t_i} + X_{HH_i}$ can be rewritten as the concatenation of X_{HH_i} with itself, i.e. $X_{HH_i} 2^{t_i} + X_{HH_i} = X_{HH_i} || X_{HH_i}$. So, the final result is

$$\langle X \rangle_{(2^n - 2^{t_i} - 1)} = \langle X_{HH_i} || X_{HH_i} + X_{HL_i} + X_H + X_L \rangle_{(2^n - 2^{t_i} - 1)}. \tag{3.14}$$

The modular reduction of $0 \le X \le 2^{2n}$ can be calculated at the cost of one 4-input n-bit CSA (Carry Save Adder). Compared to Barrett reduction method there is a considerable saving in hardware resources and timing at the same time.

3.5.2 Implementation of RNS Multipliers

RNS multiplication is implemented by a simple unsigned 66-bit multiplier for each modulus channel. The 132-bit result then will be reduced to 66 bits using efficient hardware reduction. The Xilinx series 7 FPGAs built-in DSP48E modules that include embedded 18×25-bit signed multipliers as depicted in Figure 2.7. To implement a 66×66-bit unsigned multiplication, it

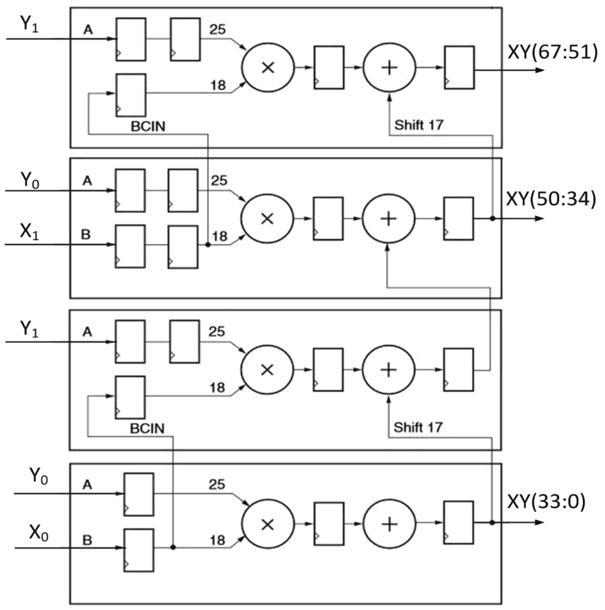


Figure 3.5: Implementation of (3.17) using 4 DSP48E

is required to use Karatsuba-Ofman algorithm [102]. In its original form, Karatsuba-Ofman algorithm splits two $2k$-bit integers X and Y into k-bit integers X_1, X_0, Y_1, and Y_0 such that

$$\begin{aligned} X &= 2^k X_1 + X_0, \\ Y &= 2^k Y_1 + Y_0. \end{aligned} \tag{3.15}$$

Then, the multiplication XY is calculated by

$$\begin{aligned} XY &= 2^{2k} Z_2 + 2^k Z_1 + Z_0, \\ Z_2 &= X_1 Y_1, \\ Z_1 &= X_1 Y_0 + X_0 Y_1, \\ Z_0 &= X_0 Y_0, \end{aligned} \tag{3.16}$$

which requires four multiplication. It can be done in parallel using four $k \times k$-bit multiplier resources in hardware. For example, if X and Y are two 34-bit integers then

$$\begin{aligned} X &= 2^{17} X_1 + X_0, \\ Y &= 2^{17} Y_1 + Y_0, \\ XY &= 2^{34} X_1 Y_1 + 2^{17}(X_1 Y_0 + X_0 Y_1) + X_0 Y_0. \end{aligned} \tag{3.17}$$

This multiplication can be done using four DSP48E units as shown in Figure 3.5. The 25×18-bit DSP48E multipliers are configured as 17×17-bit unsigned multipliers. The built-in adder and 17-bit shift right are used to implement shift and add the products. It is possible to reduce the number of multiplications to three by replacing the Z_1 to

$$Z_1 = (X_0 - X_1)(Y_1 - Y_0) + Z_0 + Z_2. \tag{3.18}$$

However, Z_1 must be calculated when the results of Z_0 and Z_2 are ready.

It is possible to decompose larger integers into more splits. For instance, if X and Y are $4k$-bit integers, then splitting X and Y into four k-bit chunks are

$$\begin{aligned} X &= 2^{3k} X_3 + 2^{2k} X_2 + 2^k X_1 + X_0, \\ Y &= 2^{3k} Y_3 + 2^{2k} Y_2 + 2^k Y_1 + Y_0, \end{aligned} \tag{3.19}$$

and multiplication is computed using

$$\begin{aligned}XY = {} & 2^{6k} X_3 Y_3 \\ & + 2^{5k}(X_2Y_3 + X_3Y_2) \\ & + 2^{4k}(X_3Y_1 + X_2Y_2 + X_1Y_3) \\ & + 2^{3k}(X_3Y_0 + X_2Y_1 + X_1Y_2 + X_0Y_3) \\ & + 2^{2k}(X_2Y_0 + X_1Y_1 + X_0Y_2) \\ & + 2^{k}(X_1Y_0 + X_0Y_1) \\ & + X_0Y_0.\end{aligned} \tag{3.20}$$

This multiplication needs 16 $k \times k$-bit multipliers. Our 66-bit RNS channels integers can be split into three 17-bit and one 15-bit integer and multiplication can be realised by 16 DSP48E modules. If we compute the products

$$\begin{aligned}P_{33} &= X_3Y_3 \\ P_{22} &= X_2Y_2 \\ P_{11} &= X_1Y_1 \\ P_{00} &= X_0Y_0 \\ P_{32} &= (X_3 - X_2)(Y_3 - Y_2) \\ P_{31} &= (X_3 - X_1)(Y_3 - Y_1) \\ P_{30} &= (X_3 - X_0)(Y_3 - Y_0) \\ P_{21} &= (X_2 - X_1)(Y_2 - Y_1) \\ P_{20} &= (X_2 - X_0)(Y_2 - Y_0) \\ P_{10} &= (X_1 - X_0)(Y_1 - Y_0) \\ S_{32} &= (P_{33} + P_{22}) \\ S_{10} &= (P_{11} + P_{00}),\end{aligned} \tag{3.21}$$

then, it is possible to do the multiplication using only ten multipliers as

$$\begin{aligned}XY = {} & 2^{6k} P_{33} \\ & + 2^{5k}(S_{32} - P_{32}) \\ & + 2^{4k}(S_{32} + P_{11} - P_{31}) \\ & + 2^{3k}(S_{32} + S_{10}P_{30} - P_{21}) \\ & + 2^{2k}(S_{10} + P_{22} - P_{20}) \\ & + 2^{k}(S_{10} - P_{10}) \\ & + P_{00}.\end{aligned} \tag{3.22}$$

In the case of our 66-bit RNS channels multiplication, either the multiplicand or the multiplier can be split into 24-bit integers and the other into 17-bit integers.

$$\begin{aligned}X &= 2^{51}X_3 + 2^{34}X_2 + 2^{17}X_1 + X_0, \\ Y &= 2^{48}Y_2 + 2^{24}Y_1 + Y_0 \\ XY &= 2^{99}X_3Y_2 + 2^{82}X_2Y_2 + 2^{65}X_1Y_2 + 2^{48}X_0Y_2 \\ &\quad + 2^{75}X_3Y_1 + 2^{58}X_2Y_1 + 2^{41}X_1Y_1 + 2^{24}X_0Y_1 \\ &\quad + 2^{51}X_3Y_0 + 2^{34}X_2Y_0 + 2^{17}X_1Y_0 + X_0Y_0.\end{aligned} \tag{3.23}$$

The 66×66-bit multiplication can be performed using twelve 18×25-bit DSP multipliers. However, extra shifts and additions are required to obtain the final result, which costs extra FPGA LUTs to implement the multiplier and increases the overall latency. The results of Two RNS channel multiplier implementations utilising 12 and 16 DSP slices on different FPGAs are presented in Table 3.3. The 66-bit RNS channel multiplier with 16 DSP48E slices shows the lowest latency. The modular multiplication is the most critical unit of the design and greatly impacts the performance of the whole system. Thus, we use the modular multiplication design with 16 DSP units for all implementations reported in this work.

Table 3.3: Implementation of different architectures of RNS channel multiplication

Device	**Max. Logic delay**	**Area**
	(ns)	LUTs / DSP / FF
ARTIX 7	16.206	513 / 16 / 0
ARTIX 7	17.420	1658/ 12 / 0
VIRTEX 7	11.525	513 / 16 / 0
VIRTEX 7	12.418	1658 / 12 / 0
KINTEX 7	11.964	513 / 16 / 0
KINTEX 7	12.013	1658 / 12 / 0
VIRTEX UltraScale+	5.910	513 / 16 / 0
VIRTEX UltraScale+	6.510	1661 / 12 / 0
KINTEX UltraScale+	5.789	513 / 16 / 0
KINTEX UltraScale+	6.489	1661 / 12 / 0

3.6 Improving RNS Montgomery modular reduction

The RNS Montgomery modular reduction (MMR) introduced in Section 2.1.2 is the basic algorithm to perform a modular reduction operation in the context of RNS. Some variants of the RNS MMR algorithm have been introduced in the literature. Guillermin proposed a method in [103] to reduce the number of multiplications in the basic MMR algorithm by merging the constants where possible. For an N-channel residue number system, Guillerman's algorithm needs $\frac{N^2+3N}{2}$ multiplications in comparison to the basic RNS MMR that needs $\frac{N^2+5N}{2}$ multiplications. Applying a special set of RNS bases called "*quadratic residues*", Kawamura et al. [104] proposed two variants of the RNS Montgomery reduction algorithm, in which the total number of multiplications is slashed to $\frac{N^2+N}{2}$. The drawback of their algorithms is the need for special moduli set and more constants for calculations; that requires a bigger ROM in hardware implementation. Similar to Guillermin's work [103], we tried to reduce the number of multiplications by merging the constants in basic RNS MMR Algorithm 5. Our proposed algorithm can be used with any RNS bases while the number of multipliers is the same as Kawamura's algorithms.

For an N-channel RNS, we proceeded to split the moduli set into half ($L = \frac{N}{2}$). In our case, the RNS moduli set $\mathcal{B}$ is split into two subsets $\mathcal{K} : \{k_1, \ldots, k_4\} = \{m_1, \ldots, m_4\}$, and

$\mathcal{Q}:\{q_1,\ldots,q_4\}=\{m_5,\ldots,m_8\}$, such that

$$K=\prod_{i=1}^{L}k_i,\ K_i=\frac{K}{k_i},\ {K_i}^{-1}=K_i \mod k_i,$$
$$Q=\prod_{i=1}^{L}q_i,\ Q_i=\frac{Q}{q_i},\ {Q_i}^{-1}=Q_i \mod q_i. \tag{3.24}$$

The RNS of integer X in base $\mathcal{K}$ is denoted as X_K and in base $\mathcal{Q}$ as X_Q. That is, $X_K=\{\langle X\rangle_{k_1},\ldots,\langle X\rangle_{k_L}\}$ and $X_Q=\{\langle X\rangle_{q_1},\ldots,\langle X\rangle_{q_L}\}$.
The RNS of X in both bases, that is, $\mathcal{B}=\mathcal{K}\cup\mathcal{Q}$, is denoted as $X_{K\cup Q}$. We introduce constants

$$P_Q=\{\langle -p^{-1}\rangle_{q_1},\ldots,\langle -p^{-1}\rangle_{q_L}\},$$
$$\Gamma_Q=\{\langle {Q_1}^{-1}\cdot\langle -p^{-1}\rangle_{q_1}\rangle_{q_1},\ldots,{Q_L}^{-1}\cdot\langle\langle -p^{-1}\rangle_{q_L}\},$$
$$\theta_{ij}=Q_i \mod k_j\ \ \forall i,j\in\{1,\ldots,L\},$$
$$\Theta_{ij}={Q^{-1}}_{k_i}\cdot p_{k_i}\cdot\theta_{ij} \mod k_j\ \ \forall i,j\in\{1,\ldots,L\}, \tag{3.25}$$
$$\Omega_{ij}=K_i \mod q_j\ \ \forall i,j\in\{1,\ldots,L\},$$
$$R_{K\cup Q}={Q^{-1}}_K\cup\Gamma_Q.$$

Then, we define $S_{K\cup Q}=X_{K\cup Q}\cdot R_{K\cup Q}$. In other terms,

$$\begin{aligned}S_{K\cup Q}&=\{s_{k_1},\ldots,s_{k_L},s_{q_1},\ldots,s_{q_L}\}\\&=\{\langle X_{k_1}{Q^{-1}}_{k_1}\rangle_{k_1},\ldots\langle X_{k_L}{Q^{-1}}_{k_L}\rangle_{k_L},\langle X_{q_1}\Gamma_{q_1}\rangle_{q_1},\ldots,\langle X_{q_L}\Gamma_{q_L}\rangle_{q_L}\}.\end{aligned} \tag{3.26}$$

Our proposed RNS Montgomery reduction algorithm is shown in Algorithm 15. The calculation of $S_{K\cup Q}$ in line 1 of the algorithm is performed in parallel with a full range RNS multiplier. The set $S_{K\cup Q}$ can be obtained from moving up the multiplication by the constant ${Q^{-1}}_K$ in line 4 of Algorithm 5 and distribute it to lines 1 and 3. The multiplications by $-p_Q^{-1}$ and p_K are merged to the base exchange $\mathcal{Q}\longrightarrow\mathcal{K}$ coefficients as well. The result in base $\mathcal{K}$, i.e. Z_K is obtained at line 3. A $\mathcal{K}\longrightarrow\mathcal{Q}$ base exchange is performed at lines 4, 5, and 6 to retrieve Z_Q from Z_K. Figure 3.6a illustrates the proposed RNS Montgomery reduction flow diagram. The "*Extended Base Exchange*" in Figure 3.6a refers to the modified coefficients of the original base exchange algorithm. Compare to the basic RNS MMR shown in Figure 2.1, three RNS multiplication stages are removed in the proposed method. The constant $R_{K\cup Q}$ is comparable to K^2 in Kawamura's dQ-RNS and sQ-RNS algorithms [104], except that we do not apply any conditions on R. That is, unlike Kawamura's algorithms, our RNS base is not required to be in any specific form. Analogous to [104], if we take out $S_{K\cup Q}$ from Algorithm 15, the total number of multiplications will be $\frac{N^2+N}{2}$, which is equal to the multiplication counts in Kawamura's dQ-RNS and sQ-RNS algorithms. This modification is shown in Algorithm 16. Since the value of α in lines 2 and 5 of Algorithm 15 is in the range $[0,L-1]$, instead of performing two extra multiplications, we have pre-computed all possible values of $-\alpha p_K$ and $-\alpha K_Q$ and saved them in the ROM. The required number of memory words to save these constants is $2(L-1)$. Table 3.4 compares our proposed MMR algorithm with the algorithms proposed by Guillermin [103], Gardino [105], and Kawamura [104] in terms of the number of multipliers and the memory words for saving constants. The number of memory words required for our proposed algorithm is $4L^2 - L$. For $N \leq 12$, this is less memory compared to other algorithms. We assumed that the input of the algorithm is (3.26), which corresponds to K^2x for sQ-RNS and dQ-RNS algorithms in [104].

Algorithm 15: Optimised RNS Montgomery algorithm

```
Input: $X_{K\cup Q}$
Output: $Z_{K\cup Q} = \{X.Q^{-1} \mod p\}_{K\cup Q}$
Constants: $\Gamma_Q$, $K_i^{-1}$, $\Theta_{ij}$, $\Omega_{ij}$ $\forall i, j \in \{1, \dots, L\}$, and $(-\alpha p)_K$, $(-\alpha K)_Q$
           $\forall \alpha \in \{1, \dots, L-1\}$.
1 $\begin{bmatrix} s_{q_1} \\ \vdots \\ s_{q_L} \end{bmatrix} \leftarrow \begin{bmatrix} \Gamma_{q_1} & \dots & 0 \\ \vdots & \Gamma_{q_i} & \vdots \\ 0 & \dots & \Gamma_{q_L} \end{bmatrix} \begin{bmatrix} X_{q_1} \\ \vdots \\ X_{q_L} \end{bmatrix}, \begin{bmatrix} s_{k_1} \\ \vdots \\ s_{k_L} \end{bmatrix} \leftarrow \begin{bmatrix} {Q^{-1}}_{k_1} & \dots & 0 \\ \vdots & {Q^{-1}}_{k_i} & \vdots \\ 0 & \dots & {Q^{-1}}_{k_L} \end{bmatrix} \begin{bmatrix} X_{q_1} \\ \vdots \\ X_{q_L} \end{bmatrix}.$
2 $\alpha \leftarrow \left\lfloor \frac{1}{2^q} \left( \sum_{i=1}^{L} \left\lfloor \frac{s_{q_i}}{2^{n-q}} \right\rfloor + 2^q\Delta \right) \right\rfloor.$
3 $\begin{bmatrix} Z_{k_1} \\ \vdots \\ Z_{k_L} \end{bmatrix} \leftarrow \begin{bmatrix} \Theta_{11} & \dots & \Theta_{1L} \\ \vdots & \Theta_{ij} & \vdots \\ \Theta_{L1} & \dots & \Theta_{LL} \end{bmatrix} \begin{bmatrix} s_{q_1} \\ \vdots \\ s_{q_L} \end{bmatrix} + \begin{bmatrix} (-\alpha p)_{k_1} \\ \vdots \\ (-\alpha p)_{k_L} \end{bmatrix} + \begin{bmatrix} s_{k_1} \\ \vdots \\ s_{k_L} \end{bmatrix}.$
4 $\begin{bmatrix} t_{k_1} \\ \vdots \\ t_{k_L} \end{bmatrix} \leftarrow \begin{bmatrix} {K_1}^{-1} & \dots & 0 \\ \vdots & {K_i}^{-1} & \vdots \\ 0 & \dots & {K_L}^{-1} \end{bmatrix} \begin{bmatrix} Z_{k_1} \\ \vdots \\ Z_{k_L} \end{bmatrix}.$
5 $\alpha \leftarrow \left\lfloor \frac{1}{2^q} \left( \sum_{i=1}^{L} \left\lfloor \frac{t_{k_i}}{2^{n-q}} \right\rfloor + 2^q\Delta \right) \right\rfloor.$
6 $\begin{bmatrix} Z_{q_1} \\ \vdots \\ Z_{q_L} \end{bmatrix} \leftarrow \begin{bmatrix} \Omega_{11} & \dots & \Omega_{1L} \\ \vdots & \Omega_{ij} & \vdots \\ \Omega_{L1} & \dots & \Omega_{LL} \end{bmatrix} \begin{bmatrix} t_{k_1} \\ \vdots \\ t_{k_L} \end{bmatrix} + \begin{bmatrix} -\alpha K_{q_1} \\ \vdots \\ -\alpha K_{q_L} \end{bmatrix}.$
```

3.6.1 Hardware implementation of the Modified Montgomery reduction algorithm

The proposed RTL architecture of Algorithm 16 is shown in Figure 3.8. It contains a sequencer state machine that provides control signals for the registers and multiplexers. The RNS multiplier uses 16 DSP slices per channel. The RNS adder uses the architecture in Figure 3.1b. The reduction to base $\mathcal{K}$ or $\mathcal{Q}$ is controlled by bit C_0 which also controls the output contents of ROM B by setting the most significant bit of the address bus. The constants Θ_{ij}, K_i^{-1}, and Ω_{ij} are saved in ROM A. The state machine controls the ROM A address bus ADR. The values of $(-\alpha p)_K$ and $(-\alpha K)_Q$ are saved in ROM B. The two-bit output of the α calculator is connected to the lower address bus of ROM B. The RNS adder and register Q4 form an accumulator. The register Q3 is a one stage pipeline for the RNS multiplier output. Registers Q1 and Q2 are used to hold data. The value of α is calculated by a simple 4-input 4-bit CSA adder depicted in figure 3.7. For both bases $\mathcal{K}$ and $\mathcal{Q}$ the $\Delta = 2^{-4}$ and $q = 2^4$. Thus, $2^q\Delta = 1$. Addition to 2^4

Algorithm 16: Modification of Algorithm 15

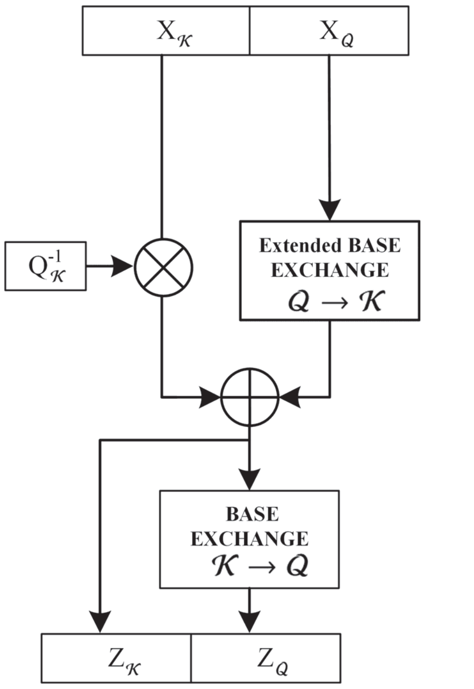
```
**Input:** $S_{K\cup Q} = X_{K\cup Q} \cdot R_{K\cup Q}$.
**Output:** $Z_{K\cup Q} = \{X.Q^{-1} \mod p\}_{K\cup Q}$.
**Constants:** ${K_i}^{-1}$, $\Theta_{ij}$, $\Omega_{ij}$ $\forall i, j \in \{1, ..., L\}$, and $(-\alpha p)_K$, $(-\alpha K)_Q$ $\forall \alpha \in \{1, ..., L-1\}$.
1 $\alpha \leftarrow \left\lfloor \frac{1}{2^q}\left(\sum_{i=1}^{L} \left\lfloor \frac{s_{q_i}}{2^{n-q}} \right\rfloor + 2^q\Delta\right)\right\rfloor$.
2 $\begin{bmatrix} Z_{k_1} \\ \vdots \\ Z_{k_L} \end{bmatrix} \leftarrow \begin{bmatrix} \Theta_{11} & \dots & \Theta_{1L} \\ \vdots & \Theta_{ij} & \vdots \\ \Theta_{L1} & \dots & \Theta_{LL} \end{bmatrix} \begin{bmatrix} s_{q_1} \\ \vdots \\ s_{q_L} \end{bmatrix} + \begin{bmatrix} (-\alpha p)_{k_1} \\ \vdots \\ (-\alpha p)_{k_L} \end{bmatrix} + \begin{bmatrix} s_{k_1} \\ \vdots \\ s_{k_L} \end{bmatrix}$.
3 $\begin{bmatrix} t_{k_1} \\ \vdots \\ t_{k_L} \end{bmatrix} \leftarrow \begin{bmatrix} {K_1}^{-1} & \dots & 0 \\ \vdots & {K_i}^{-1} & \vdots \\ 0 & \dots & {K_L}^{-1} \end{bmatrix} \begin{bmatrix} Z_{k_1} \\ \vdots \\ Z_{k_L} \end{bmatrix}$.
4 $\alpha \leftarrow \left\lfloor \frac{1}{2^q}\left(\sum_{i=1}^{L} \left\lfloor \frac{t_{k_i}}{2^{n-q}} \right\rfloor + 2^q\Delta\right)\right\rfloor$.
5 $\begin{bmatrix} Z_{q_1} \\ \vdots \\ Z_{q_L} \end{bmatrix} \leftarrow \begin{bmatrix} \Omega_{11} & \dots & \Omega_{1L} \\ \vdots & \Omega_{ij} & \vdots \\ \Omega_{L1} & \dots & \Omega_{LL} \end{bmatrix} \begin{bmatrix} t_{k_1} \\ \vdots \\ t_{k_L} \end{bmatrix} + \begin{bmatrix} -\alpha K_{q_1} \\ \vdots \\ -\alpha K_{q_L} \end{bmatrix}$.
```

Table 3.4: Comparing proposed MMR algorithm performance with other works

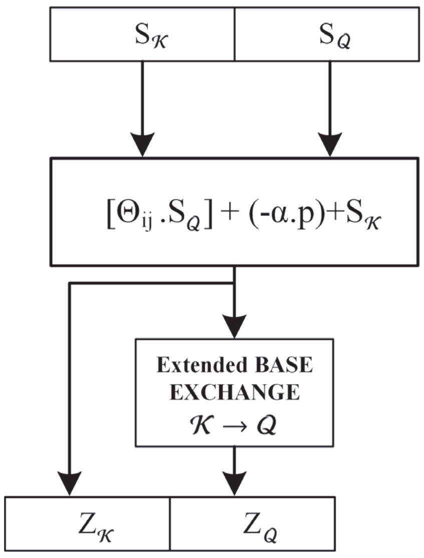
Algorithm	RNS Multipliers	Required Memory	RNS Multipliers / Required memory for $N = 8$
Proposed	$2L^2 + L$	$4L^2 - L$	33 / 60
[104] sQ	$2L^2 + L$	$2L^2 + 12L + 1$	33 / 81
[104] dQ	$2L^2 + L$	$2L^2 + 13L + 1$	33 / 85
[105]	$2L^2 + 4L$	$2L^2 + 11L$	36 / 76
[103]	$2L^2 + 5L$	$2L^2 + 11L$	37 / 76

Table 3.5: Hardware implementation of the proposed RNS Montgomery reduction

Algorithm	K-LUT / DSP / FF	Max. frequency	Clock cycles
Proposed	3921 / 64/ 1114	188.67	20
[104] sQ	4076 / 84 / 2104	139.5	15
[104] dQ	4247 / 84 / 2329	142.7	18


$X_{\mathcal{K}}$
$X_{\mathcal{Q}}$
$Q^{-1}_{\mathcal{K}}$
Extended BASE EXCHANGE $\mathcal{Q} \rightarrow \mathcal{K}$
BASE EXCHANGE $\mathcal{K} \rightarrow \mathcal{Q}$
$Z_{\mathcal{K}}$
$Z_{\mathcal{Q}}$

(a) Optimised RNS Montgomery reduction algorithm.


$S_{\mathcal{K}}$
$S_{\mathcal{Q}}$
$[\Theta_{ij}.S_{\mathcal{Q}}] + (-\alpha.p) + S_{\mathcal{K}}$
Extended BASE EXCHANGE $\mathcal{K} \rightarrow \mathcal{Q}$
$Z_{\mathcal{K}}$
$Z_{\mathcal{Q}}$

(b) Modified RNS Montgomery reduction algorithm.

Figure 3.6: Flow diagram of two Modifications on the RNS Montgomery reduction algorithm.

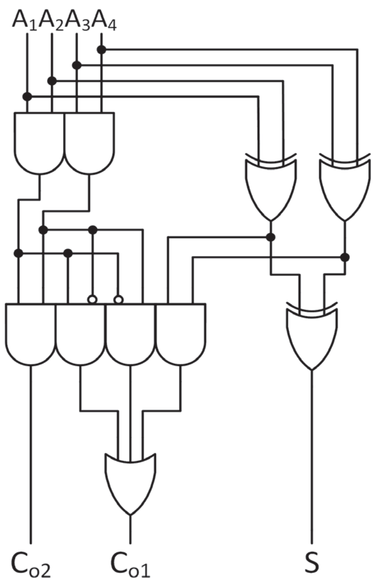


Figure 3.7: 4-input CSA adder circuit

can be simply obtained by concatenating '01' at the end of S, as shown in (3.27). Then, α is the last two most significant bits of L.

$$\begin{aligned}
P_0 &= A_1 \oplus A_2\\
P_1 &= A_3 \oplus A_4\\
G_0 &= A_1 \wedge A_2\\
G_1 &= A_3 \wedge A_4\\
S &= (P_0 \oplus P_1)\\
C_0 &= ((P_0 \wedge P_1) \vee (\overline{G_0} \wedge G1) \vee (G_0 \wedge \overline{G_1}))\\
C_1 &= (G_0 \wedge G_1)\\
L &= (\text{`01'}||S) + (\text{`0'}||C_0||\text{`0'}) + (C_1||\text{`00'})
\end{aligned} \tag{3.27}$$

The state machine consists of 20 states that control the flow of data. Figure 3.9 shows the operations at each clock cycle. When the state machine is in reset mode, all control bits are

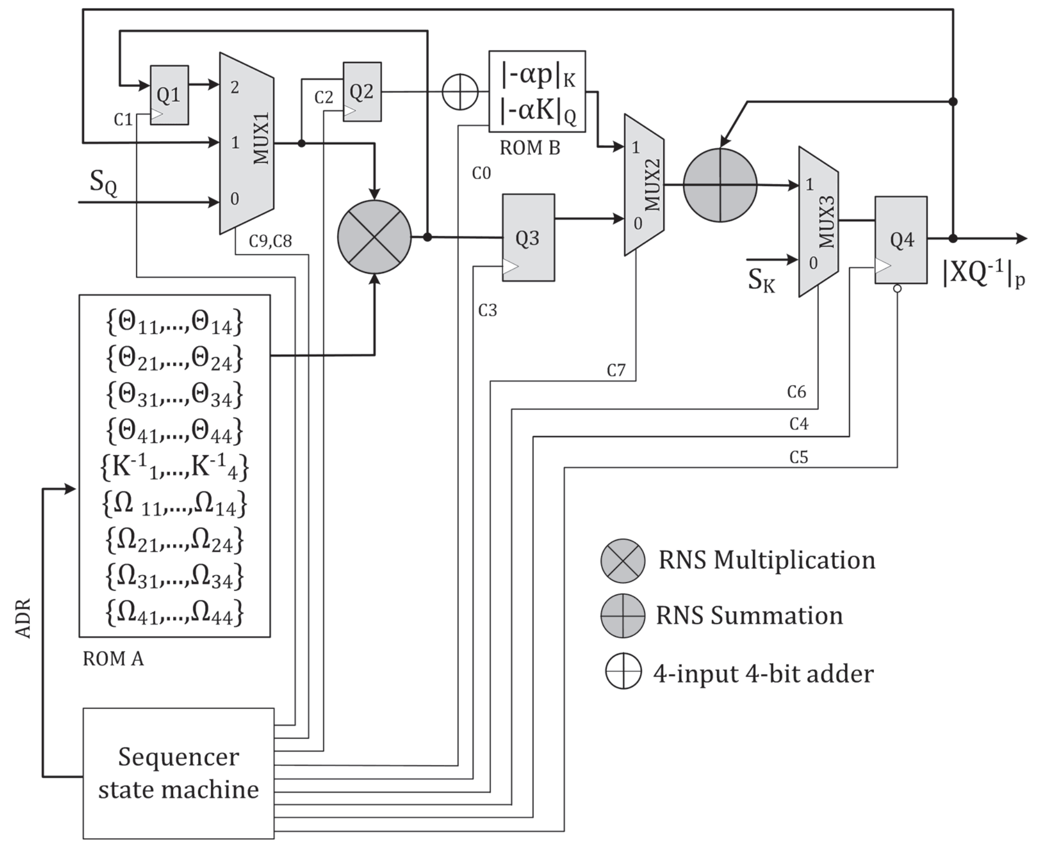


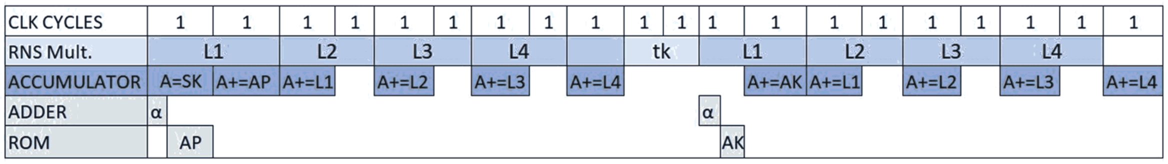
Figure 3.8: Proposed hardware for modified RNS Montgomery reduction Algorithm 16.

set to zero. It is expected that valid data is provided at the inputs before exiting from the reset state. The activities at each state are outlined as follows:

S_0: It is assumed that the values of s_Q and s_K are valid after exit from a reset state. At this state, s_Q and s_K are loaded to Q2 and Q4 registers, respectively. The Address of ROM A is '0000'.

S_1: The value of $-\alpha p$ is fetched from ROM B during the previous state and is added to the accumulator at the rising edge of C_4. The contents of the accumulator (the output of Q4) is now $\langle s_K - \alpha p\rangle_K$.

S_2: The RNS multiplier's output is latched to Q3 at the rising edge of C_3. The Address of the ROM A is incremented to '0001'.
S_3: The accumulator is updated by adding the output of the RNS multiplier.
$S_4 - S_8$: The product of s_Q and the corresponding row of matrix Θ, i.e. Θ_i , $\forall i \in \{2,3,4\}$, is calculated and added to the accumulator.
S_9: At the previous state, S_8, the address of ROM A has been updated to '0100', which is pointing to the values of K^{-1}. The RNS multiplier performs step 3 of Algorithm 16.
S_{10}: The result of the RNS multiplier is ready and is latched by Q1 at the rising edge of C_1. ADR is updated to '0101'.
S_{11}: At the rising edge of C_2, the output of Q1 is loaded to Q2 to calculate the value of $-\alpha K$. Signal C_0 is set to '1'; indicating that the modular reductions will be done in base Q to the end of the states. Signal C_0 is the MSB of the ROM B address which points to the second partition of RAM B where the values of $-\alpha K$ are saved. The accumulator is reset to zero by setting C_5 to '1' for one clock cycle.
S_{12}: The value of $-\alpha K$ is added to the accumulator. The RNS multiplier output is latched at the rising edge of C_2. ADR is incremented to '0110'.
S_{13}: The output of ROM B ($-\alpha K$) is added to the accumulator at the rising edge of C_4.
S_{14}: Accumulator is updated by the output of Q3. ADR is incremented to '0111'.
$S_{15} - S_{18}$: The RNS multiplier completes the multiplication of matrix Ω rows to the corresponding t_k (step 5 of Algorithm 16). The accumulator is updated by the result of the last multiplication.
S_{19}: At the rising edge of C_4, the final result is set to the output of the circuit.



CLK CYCLES	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
RNS Mult.	L1		L2		L3		L4			tk		L1		L2		L3		L4		
ACCUMULATOR	A=SK	A+=AP	A+=L1		A+=L2		A+=L3		A+=L4				A+=AK	A+=L1		A+=L2		A+=L3		A+=L4
ADDER	α											α								
ROM	AP											AK								

Figure 3.9: Data flow of modified RNS Montgomery reduction (Algorithm 16) per clock cycle.

3.7 The Sum of residues modular reduction[1]

RNS Modular reduction based on the sum of residues (SOR) algorithm was first presented by Phillips et al. [107] in 2010. The SOR algorithm hardware implementation was proposed later in [97]. The disadvantage of the Phillips' SOR algorithm is that unlike the Montgomery reduction method, the output is an unknown and variable multiple of the "$X \mod p$" value. Although this algorithm offers a high level of parallelism in calculations, the proposed implementation in [97] has a considerably big area.

We propose an improvement to the sum of residues algorithm by introducing the correction factor κ to obtain a precise result. We also present a new design to improve the area in comparison to [97].

Suppose, $Z < M$ is an integer and $p < M^{\frac{1}{2}}$ is a prime. Based on the CRT, Z can be

[1]This section is a part of the published work [106]

presented using equation (2.14) by replacing X to Z. Reducing Z modulo p yields

$$Z \mod p = \langle Z\rangle_p = \langle\sum_{i=1}^{N}\gamma_i M_i\rangle_p - \langle\alpha M\rangle_p, \tag{3.28}$$

where $\gamma_i = \langle z_i M_i^{-1}\rangle_{m_i}$ can be easily performed by an unsigned integer $n \times n$ multiplier and a modular reduction detailed in Section 3.5.1. Calculation of α is outlined in Section 2.1.2. Let's introduce a new integer V as:

$$V = \sum_{i=1}^{N}\gamma_i\langle M_i\rangle_p - \langle\alpha M\rangle_p. \tag{3.29}$$

Recalling the fact that for any integer Z we can find an integer κ such that $\langle Z\rangle_p = Z - \kappa\cdot p$, it can be investigated that the difference of V and $\langle Z\rangle_p$ is a multiple of modulus p.

$$\begin{aligned} V - \langle Z\rangle_p &= \sum_{i=1}^{N}\gamma_i\langle M_i\rangle_p - \langle\sum_{i=1}^{N}\gamma_i M_i\rangle_p \\ &= \sum_{i=1}^{N}\gamma_i(M_i - \nu\cdot p) - \sum_{i=1}^{N}\gamma_i M_i - \mu\cdot p \\ &= (\sum_{i=1}^{N}\gamma_i.\nu - \mu)\cdot p \\ &= \kappa\cdot p. \end{aligned} \tag{3.30}$$

ν and μ are constants such that $\langle M_i\rangle_p = M_i - \nu\cdot p$, and $\langle\sum_{i=1}^{N}\gamma_i M_i\rangle_p = \sum_{i=1}^{N}\gamma_i M_i - \mu\cdot p$.

The factor (κ) is a function of γ_i, not a constant. Therefore, the value of V, which is actually the output of SOR algorithm introduced in [107] and [97], is not presenting the true reduction of $\langle Z\rangle_p$. In other words,

$$V = \kappa\cdot p + \langle Z\rangle_p = \sum_{i=1}^{N}\gamma_i\langle M_i\rangle_p + \langle -\alpha M\rangle_p. \tag{3.31}$$

The values of $\langle M_i\rangle_p$ and $\langle\alpha M\rangle_p$ for $\alpha \in \{0, 1, \cdots, N-1\}$ are known and can be implemented in hardware as pre-computed constants.
The RNS form of V resulted from (3.29) is

$$\begin{bmatrix}\langle V\rangle_{m_1}\\ \langle V\rangle_{m_2}\\ \vdots\\ \langle V\rangle_{m_N}\end{bmatrix} = \left(\sum_{i=1}^{N}\gamma_i\begin{bmatrix}\langle\langle M_i\rangle_p\rangle_{m_1}\\ \langle\langle M_i\rangle_p\rangle_{m_2}\\ \vdots\\ \langle\langle M_i\rangle_p\rangle_{m_N}\end{bmatrix}\right) + \begin{bmatrix}\langle -\alpha\langle M\rangle_p\rangle_{m_1}\\ \langle -\alpha\langle M\rangle_p\rangle_{m_2}\\ \vdots\\ \langle -\alpha\langle M\rangle_p\rangle_{m_N}\end{bmatrix}. \tag{3.32}$$

If (3.32) deducted by $\{\langle\kappa\cdot p\rangle_{m_1}, \langle\kappa\cdot p\rangle_{m_2}, \cdots, \langle\kappa\cdot p\rangle_{m_N}\}$, the accurate value of Z_p in RNS will be obtained.

$$\begin{bmatrix}\langle Z_p\rangle_{m_1}\\ \langle Z_p\rangle_{m_2}\\ \vdots\\ \langle Z_p\rangle_{m_N}\end{bmatrix} = \left(\sum_{i=1}^{N}\gamma_i\begin{bmatrix}\langle\langle M_i\rangle_p\rangle_{m_1}\\ \langle\langle M_i\rangle_p\rangle_{m_2}\\ \vdots\\ \langle\langle M_i\rangle_p\rangle_{m_N}\end{bmatrix}\right) + \begin{bmatrix}\langle -\alpha\langle M\rangle_p\rangle_{m_1}\\ \langle -\alpha\langle M\rangle_p\rangle_{m_2}\\ \vdots\\ \langle -\alpha\langle M\rangle_p\rangle_{m_N}\end{bmatrix} - \begin{bmatrix}\langle\kappa.p\rangle_{m_1}\\ \langle\kappa.p\rangle_{m_2}\\ \vdots\\ \langle\kappa.p\rangle_{m_N}\end{bmatrix}. \tag{3.33}$$

3.7.1 Calculation of κ

Dividing each side of (3.28) by the modulus p yields:

$$\kappa + \frac{\langle Z \rangle_p}{p} = \sum_{i=1}^{N} \frac{\gamma_i \langle Mi \rangle_p}{p} + \frac{\langle -\alpha M \rangle_p}{p}. \tag{3.34}$$

The coefficient κ is an integer. Reminding that $\frac{\langle Z \rangle_p}{p} < 1$ and $\frac{\langle -\alpha M \rangle_p}{p} < 1$, κ is calculated as

$$\kappa = \left\lfloor \sum_{i=1}^{N} \frac{\langle \gamma_i M_i \rangle_p}{p} \right\rfloor. \tag{3.35}$$

The modulus p is considered to be a pseudo Mersenne prime in general form of $p = 2^W - \epsilon$ where $2^W \gg \epsilon$. For example: $p_S = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ and $p_E = 2^{255} - 19$ are the field modulus for the SECG recommended curve SECP256K1 [30] and the Twisted Edwards Curve ED25519 [40], respectively.

Substitution of fractional equation $\frac{1}{2^W - \epsilon} = \frac{1}{2^W}(1 + \frac{\epsilon}{2^W - \epsilon})$ in (3.35) results:

$$\kappa = \left\lfloor \sum_{i=1}^{N} \frac{\gamma_i \langle M_i \rangle_p}{2^W} \left(1 + \frac{\epsilon}{(2^W - \epsilon)} \right) \right\rfloor. \tag{3.36}$$

Considering that $\langle M_i \rangle_p < p$ and $\gamma_i < 2^n$, if we choose

$$\epsilon < \frac{2^{W-n}}{N}, \tag{3.37}$$

then, $\sum_{i=1}^{N} \frac{\gamma_i \langle M_i \rangle_p}{2^W} \frac{\epsilon}{(2^W - \epsilon)} < 1$, and the value of κ resulted from (3.36) is

$$\kappa = \left\lfloor \sum_{i=1}^{N} \frac{\gamma_i \langle M_i \rangle_p}{2^W} \right\rfloor. \tag{3.38}$$

The condition in (3.37) provides a new boundary for choosing the field modulus p. It is a valid condition for the most known prime moduli p used practically in cryptography. Table 3.6 shows the validity of κ for some standard curves based on (3.38).

Table 3.6: Checking validity of κ for some standard curves SEC2, ED25519.

CURVE	**Modulus** p	N	n	$\frac{2^{W-n}}{N}$	ϵ
ED25519	$2^{255}-19$	8	66	2^{186}	19
SECP160K1	$2^{160}-2^{32}-21389$	5	66	$\frac{2^{94}}{5}$	$2^{32}+21389$
SECP160R1	$2^{160}-2^{32}-1$	5	66	$\frac{2^{94}}{5}$	$2^{32}+1$
SECP192K1	$2^{192}-2^{32}-4553$	6	66	$\frac{2^{125}}{3}$	$2^{32}+4553$
SECP192R1	$2^{192}-2^{64}-1$	6	66	$\frac{2^{125}}{3}$	$2^{64}+1$
SECP224K1	$2^{224}-2^{32}-6803$	7	66	$\frac{2^{158}}{7}$	$2^{32}+6803$
SECP224R1	$2^{224}-2^{96}+1$	7	66	$\frac{2^{158}}{7}$	$2^{96}-1$
SECP256K1	$2^{256}-2^{32}-977$	8	66	2^{187}	$2^{32}+977$
SECP2384R1	$2^{384}-2^{128}-2^{96}+2^{31}-1$	12	66	$\frac{2^{316}}{3}$	$2^{128}+2^{96}-2^{31}+1$
SECP521R1	$2^{521}-1$	16	66	2^{451}	1

The hardware implementation of (3.38) needs a 66×256-bit multiplier. For an efficient hardware implementation, it is essential to avoid such a big multiplier. To compute the value of κ in hardware, we used

$$\kappa = \left\lfloor \frac{1}{2^T} \sum_{i=1}^{N} \gamma_i \left\lfloor \frac{\langle M_i \rangle_p}{2^{W-T}} \right\rfloor \right\rfloor. \tag{3.39}$$

The integer T must be selected such that the equality of (3.38) and (3.39) is guaranteed. Using a MAPLE program, we realised that $T = 72$ for SECP256K1 and $T = 71$ for ED25519 are the best solutions for area-efficient hardware. In this case, as the term $\left\lfloor \frac{\langle M_i \rangle_p}{2^{W-T}} \right\rfloor$ is 55 bits for SECP256K1 and 44 bits for ED25519, the 66×55-bit and 66×44-bit multipliers are required to compute κ, respectively.
Therefore, the coefficient of κ for SECP256K1 can be calculated efficiently by the following equation:

$$\kappa = \left\lfloor \frac{1}{2^{72}} \sum_{i=1}^{N} \gamma_i \left\lfloor \frac{\langle M_i \rangle_{p_S}}{2^{184}} \right\rfloor \right\rfloor. \tag{3.40}$$

Similarly, for ED25519, κ can be calculated using the below formula:

$$\kappa = \left\lfloor \frac{1}{2^{71}} \sum_{i=1}^{N} \gamma_i \left\lfloor \frac{\langle M_i \rangle_{p_E}}{2^{184}} \right\rfloor \right\rfloor. \tag{3.41}$$

The value of $\left\lfloor \frac{\langle M_i \rangle_p}{2^{184}} \right\rfloor$ can be pre-computed and saved in the hardware for $i = 1$ to N. The integer κ is at most 52-bit long for SECP256K1 and 42-bit long for ED25519. As a result, the RNS conversion is not required. ($\kappa_i = \kappa \mod m_i = \kappa$) and κ can be directly used in RNS calculations.

The calculation of κ can be done in parallel and will not impose an extra delay in the design. Finally, to find $z = X \mod p$ it is required to compute $\kappa \cdot \langle p \rangle_{m_i}$. Note that $\langle p \rangle_{m_i}$ is a constant and can be pre-computed as well. Therefore, it follows that

$$z_i = \left\langle \langle V \rangle_{m_i} - \langle \kappa \cdot p \rangle_{m_i} \right\rangle_{m_i}. \tag{3.42}$$

The number of operations can be reduced by pre-computing $\langle -p \rangle_{m_i}$ instead of $\langle p \rangle_{m_i}$. (A modular subtraction consists of two operations: $\forall a, b < m_i$, $\langle a - b \rangle_{m_i} = \langle a + (m_i - b) \rangle_{m_i}$). Then z_i is calculated directly by

$$z_i = \left\langle \langle V \rangle_{m_i} + \langle \kappa \cdot \langle -p \rangle_{m_i} \rangle_{m_i} \right\rangle_{m_i}. \tag{3.43}$$

Algorithm 17 presents the RNS modulo p multiplication $\{x_1, x_2, \ldots, x_N\} \times \{y_1, y_2, \ldots, y_N\} \mod p$ over moduli base $\mathcal{B}$ using the improved sum of residues method. Calculations in stages 4 and 5 are concurrent. Different levels of parallelism are possible in hardware implementation by adding one or more RNS multipliers to perform stage 5.3 calculations in parallel.

As discussed, the coefficient κ is a 52-bit(42-bit) integer for the SECP256K1(ED25519) design. Consequently, the output of the original SOR algorithm [107] represented in (3.29) is as big as 308(297) bits. In conclusion, the hardware introduced in [97], [108], and [98] cannot calculate two tandem modular multiplications while the product of the second stage inputs has a higher bit number than the dynamic range that violates the CRT. In cryptographic applications, it is generally required to do multiple modular multiplications. Our correction to the SOR algorithm ensures that the inputs of the next multiplication stage are in range.

3.7.2 The SOR algorithm hardware implementation and performance

The required memory to implement constant parameters of Algorithm 17 is $N((2N + 2)n + n')$ bits where n' is the biggest bit number of $\left\lfloor \frac{\langle M_i \rangle_p}{2^{W-T}} \right\rfloor$, $i \in \{1, \ldots, N\}$. In our case $n' = 55$ for SECP256K1 and $n' = 44$ for ED25519. Therefore, the required memory is 9944 and 9856 bits for the SECP256K1 and ED25519 respectively.

The RNS multiplier is implemented by using 16 DSP slices followed by a combinational reduction logic per channel. The total number of 128 DSP resources are used for an RNS multiplier. The logic delays of RNS adder and multiplier listed in Tables 3.2 and 3.3 determine the overall design latency and performance. The maximum RNS adder logic and routing latency are less than half of the RNS multiplier logic and net delays. The system clock cycle is chosen such that an RNS addition is complete in one clock period and an RNS multiplication result is ready in two clock periods. Figure 3.10 presents a simplified block diagram of the Algorithm 17 with non-pipelined architecture. We name this architecture as SOR_1M_N. The sequencer state machine provides select signals of the multiplexers and clocks for internal registers. The inputs of the circuit are two 256-bit integers X and Y in RNS representation over base $\mathcal{B}$; that is, $\{x_1, \ldots, x_N\}$ and $\{y_1, dots, y_N\}$, respectively.

The RNS multiplier inputs are selected by multiplexers MUX1 and MUX2. At the second clock cycle the output of multiplier, that is, $xy_i = \langle x_i \cdot y_i \rangle_{m_i}$, is latched by register $Q1$. At the fourth clock cycle, $\gamma_i = \langle xy_i \cdot M_i^{-1} \rangle_{m_i}$ is calculated and latched by register $Q1$. The calculation of α starts after the fourth clock cycle, by adding the eight most significant bits of γ_1 to γ_8 to the offset $2^q\Delta = 2^4$. The three most significant bits of the result are used to select the value of $\langle -\alpha \cdot \langle M \rangle_p \rangle_{m_i}$ from the Look up table. Figure 3.11 illustrates the hardware implementation of $\langle -\alpha \cdot \langle M \rangle_p \rangle_{m_i}$. At the next $3N$ clock cycles $\langle \gamma_i \langle M_i \rangle_p \rangle_{m_j}$ will be calculated and accumulated in

Algorithm 17: Improved Sum of residues reduction

```
Required: $p, \Delta, q, \mathcal{B} = \{m_1, \cdots, m_N\}, m_1 > m_2 > \cdots > m_N, n = \lceil \log_2 m_1 \rceil, W = \lceil \log_2 p \rceil, T,$
    $N \geq \lceil \frac{2W}{n} \rceil.$
Required: $M = \prod_{i=1}^{N} m_i, \hat{M} = (1-\Delta)M, M_i = \frac{M}{m_i}$ for $i = 1$ to $N$.
Required: Tables $\begin{bmatrix} \langle M_1{}^{-1} \rangle_{m_1} \\ \langle M_2{}^{-1} \rangle_{m_2} \\ \vdots \\ \langle M_N{}^{-1} \rangle_{m_N} \end{bmatrix}, \begin{bmatrix} \langle -p \rangle_{m_1} \\ \langle -p \rangle_{m_2} \\ \vdots \\ \langle -p \rangle_{m_N} \end{bmatrix}, \begin{bmatrix} \left\lfloor \frac{\langle M_1 \rangle_p}{2^{W-T}} \right\rfloor \\ \vdots \\ \left\lfloor \frac{\langle M_N \rangle_p}{2^{W-T}} \right\rfloor \end{bmatrix}$, and $\begin{bmatrix} \langle \langle M_i \rangle_p \rangle_{m_1} \\ \langle \langle M_i \rangle_p \rangle_{m_2} \\ \vdots \\ \langle \langle M_i \rangle_p \rangle_{m_N} \end{bmatrix}$ for $i = 1$ to $N$.
Required: Table $\begin{bmatrix} \langle \alpha \cdot \langle -M \rangle_p \rangle_{m_1} \\ \langle \alpha \cdot \langle -M \rangle_p \rangle_{m_2} \\ \vdots \\ \langle \alpha \cdot \langle -M \rangle_p \rangle_{m_N} \end{bmatrix}$ for $\alpha = 1$ to $N-1$
Input: Integers $X$ and $Y$, $0 \leq X, Y < \hat{M}$ in form of RNS: $\{x_1, \cdots, x_N\}$ and $\{y_1, \cdots, y_N\}$.
Output: Presentation of $Z = X \cdot Y \mod p$ in RNS:$\{z_1, \cdots, z_N\}$.
1. for $i = 1$ to $N$ do
    $xy_i \leftarrow \langle x_i \cdot y_i \rangle_{m_i}$.
    $\gamma_i \leftarrow \langle xy_i \langle M_i{}^{-1} \rangle_{m_i} \rangle_{m_i}$.
end
3. for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        $Y_{ij} \leftarrow \gamma_i \langle \langle M_i \rangle_p \rangle_{m_j}$.
    end
end
4. for $i = 1$ to $N$ do
    4.1 $\alpha \leftarrow \left\lfloor \frac{1}{2^q} \left( \sum_{i=1}^{N} \left\lfloor \frac{\gamma_i}{2^{n-q}} \right\rfloor + 2^q \Delta \right) \right\rfloor$.
    4.2 $\kappa \leftarrow \left\lfloor \frac{1}{2^T} \sum_{i=1}^{N} \gamma_i \left\lfloor \frac{\langle M_i \rangle_p}{2^{W-T}} \right\rfloor \right\rfloor$.
end
5. for $i = 1$ to $N$ do
    5.1 Calculate $\langle \kappa \cdot \langle -p \rangle_{m_i} \rangle_{mi}$ .
    5.2 Read $\langle \alpha \langle -M \rangle_p \rangle_{m_i}$ from the table.
    5.3 $sum_i \leftarrow \langle \sum_{j=1}^{N} Y_{ji} \rangle_{m_i}$ .
end
6. for $i = 1$ to $N$ do
    $z_i \leftarrow \langle sum_i + \alpha \langle -M \rangle_p \rangle_{m_i} + \langle \kappa \langle -p \rangle_{m_i} \rangle_{m_i}$.
end
```

register $Q2$. The RNS multiplier must be idle for one clock cycle, letting the RNS adder of the accumulator be completed and latched whenever accumulation of the results is required. The value of κ is calculated in parallel using the hardware shown in Figure 3.12. The $\langle -\kappa p \rangle_{m_i}$ is calculated at the $(3N+5)$ and $(3N+6)$ cycles and will be added to the accumulator Q2 at the last clock cycle. The sum of moduli reduction is completed in $(3N+7)$ clock cycles. Figure 3.13 shows the data flow diagram of SOR_1M_N architecture at every clock cycle.

A pipelined design is depicted in Figure 3.14. Here, an extra register Q3 latches the RNS multiplier's output. So, the idle cycles in SOR_1M_N are removed. We call this design SOR_1M_P. The data flow diagram of SOR_1M_P architecture is illustrated in Figure 3.15 The Algorithm 17 can be performed in $2(N+4)$ clock cycles using this architecture.

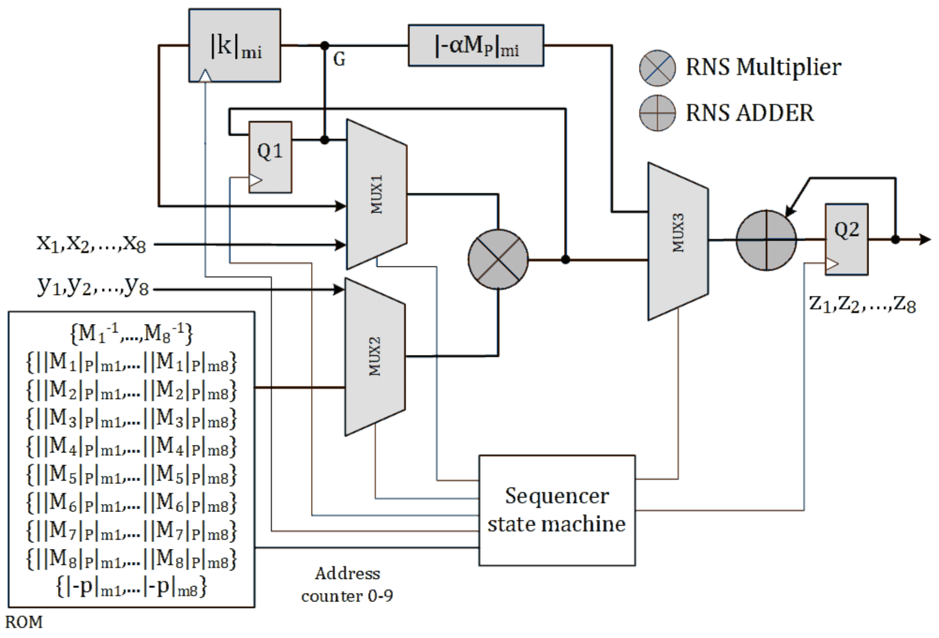


Figure 3.10: Sum of residues reduction block diagram non-pipelined (SOR_1M_N) design.

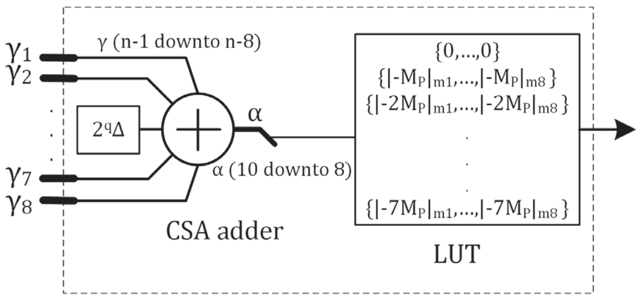


Figure 3.11: Implementation of $\langle\langle -\alpha M\rangle_p\rangle_{m_i}$.

Parallel designs are possible by adding RNS multipliers to the design. Figure 3.16 shows the architecture of using two identical RNS multipliers in parallel to implement Algorithm 17. We tag this architecture as SOR_2M. The calculation of $\langle\gamma_i\langle M_i\rangle_p\rangle_{m_j}$, $(i = 1\cdots N)$ is split between two RNS multipliers. So, the required time to calculate all N terms is halved. As shown in Figure 3.12, An extra $n\times n$ multiplier is also required to calculate κ in time. The latency of SOM_2M architecture is $2(\frac{N}{2}+5)$ clock cycles. Theoretically, the latency could be as small as 12 clock cycles using N parallel RNS multipliers. Figure 3.17 shows the data flow diagram of SOM_2M architecture.

Table 3.7, shows implementation results on VIRTEX 7, KINTEX 7, VIRTEX UltraScale+™, and KINTEX UltraScale+ FPGA series. VIVADO 2017.4 is used for VHDL codes synthesis. The fastest design is realised using KINTEX UltraScale+ that clock frequency 187.13 MHz is reachable. Table 3.7 outlines the implementation results on different platforms.

3.7.3 SOR performance

In Table 3.8, we have outlined the implementation results of recent similar works in the context of RNS. The design in [97] and [98] are based on the SOR algorithm in [107]. Both of them use forty 14-bit co-prime moduli as RNS base to provide a 560-bit dynamic range. Barrett

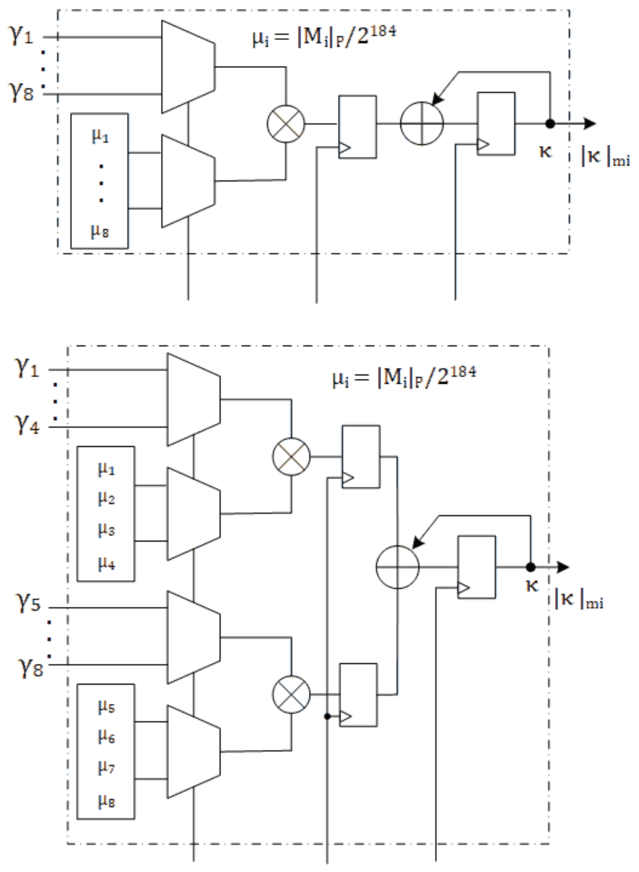


Figure 3.12: Implementation of $\langle \kappa \langle -p \rangle \rangle_{m_i}$ in architectures SOR_1M_N and SOR_1M_P (Up) and in architecture SOR_2M (Down).

CLK CYCLES	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
RNS Mult.	Z	G	L1		L2			L3			L4			L5			L6			L7			L8			KP		
ACCUMULATOR				A=A+L1			A=A+L2			A=A+L3			A=A+L4			A=A+L5			A=A+L6			A=A+L7			A=A+L8	A=A+AL		A=A+KP
MULTIPLIER			K1	K2		K3		K4		K5		K6		K7		K8												
ACCUMULATOR				K=K+K1		K=K+K2		K=K+K3		K=K+K4		K=K+K5		K=K+K6		K=K+K7		K=K+K8										
ADDER			α																									
ROM			AL																									

Figure 3.13: Data flow of SOR non–pipelined SOR_1M_N architecture.

reduction method [11] is used for moduli multiplication at each channel. Barrett reduction algorithm costs two multiplications and one subtraction; which is not an optimised method for high-speed designs. The design in [97] is a combinational logic and performs an RNS modular reduction in one clock cycle. The area for pipelined and non-pipelined architectures of this design, as reported in [98], is (34.34 KLUTs+2016 DSPs) and (36.5 KLUTs+2016 DSPs), respectively (the number of registers is not reported). The MM_SPA design in [98] is more reasonable in terms of the logic size (11.43 KLUT+512 DSPs). However, in contrast to our SOR_2M design on VIRTEX-7, it consumes more hardware resources and is considerably slower.

These designs are based on SOR algorithm in [107] that is not performing a complete reduction. As discussed in Section 3.7.1, their outputs can exceed the RNS dynamic range and give out completely incorrect results.
The authors used eight 65-bit moduli base for their RNS hardware which is similar to our design. The achieved clock frequencies for these two designs are 139.5 MHz and 142.7 MHz, respectively. The input considered for the algorithms is the RNS presentation of K^2x; where x is equivalent to Z in our notations and K^2 is a constant. A modular multiplication with sQ

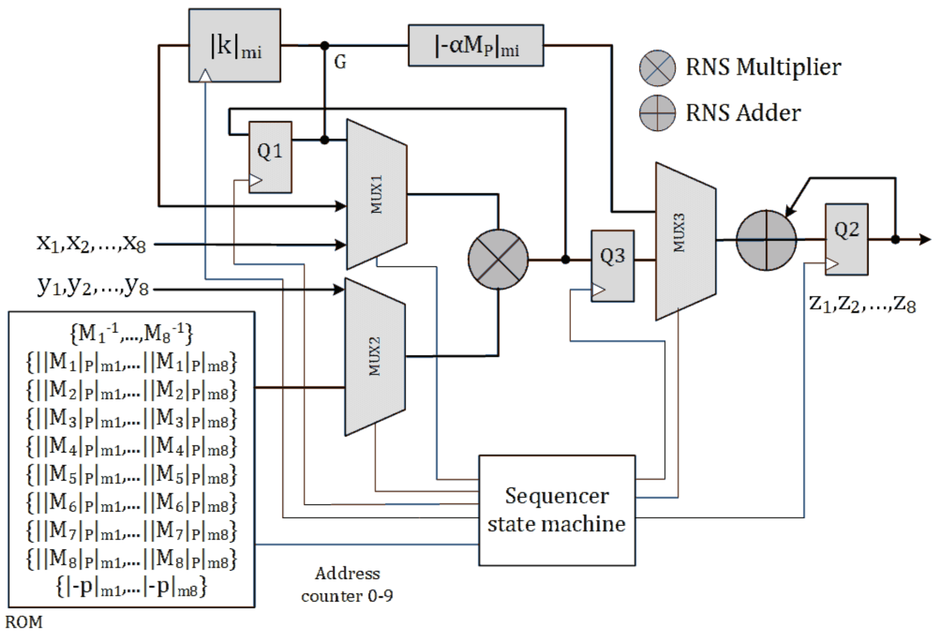


Figure 3.14: Sum of residues reduction block diagram with pipelined (SOR_1M_P) design.

CLK CYCLES	2	2	2	2	2	2	2	2	2	2	1	2	1
RNS Mult.	Z	G	L1	L2	L3	L4	L5	L6	L7	L8		KP	
ACCUMULATOR				A=A+L1	A=A+L2	A=A+L3	A=A+L4	A=A+L5	A=A+L6	A=A+L7	A=A+L8	A=A+AL	A=A+KP
MULTIPLIER			K1	K2	K3	K4	K5	K6	K7	K8			
ACCUMULATOR				K=K+K1	K=K+K2	K=K+K3	K=K+K4	K=K+K5	K=K+K6	K=K+K7	K=K+K8		
ADDER			α										
ROM			AL										

Figure 3.15: Data flow of SOR pipelined SOR_1M_P architecture

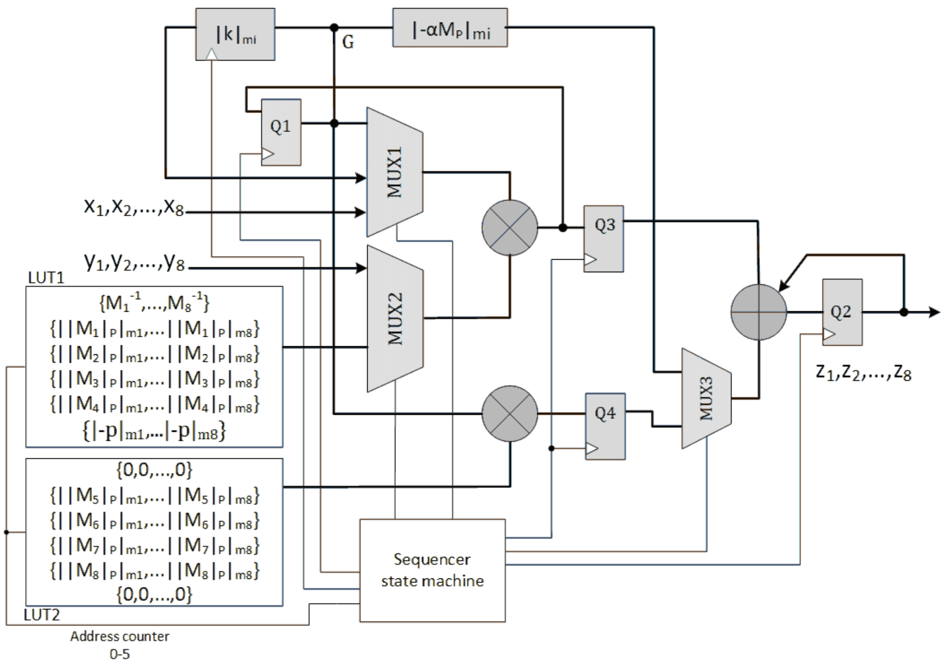
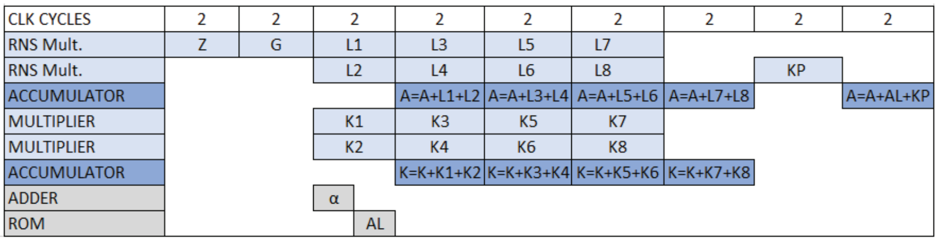


Figure 3.16: Sum of residues block diagram using two parallel pipelined (SOR_2M) design.

and dQ algorithms needs two initial RNS multiplication to provide input for the algorithm. To have a fair comparison, the latency of these multiplications must be taken into account. As



CLK CYCLES	2	2	2	2	2	2	2	2	2
RNS Mult.	Z	G	L1	L3	L5	L7			
RNS Mult.			L2	L4	L6	L8		KP	
ACCUMULATOR				A=A+L1+L2	A=A+L3+L4	A=A+L5+L6	A=A+L7+L8		A=A+AL+KP
MULTIPLIER			K1	K3	K5	K7			
MULTIPLIER			K2	K4	K6	K8			
ACCUMULATOR				K=K+K1+K2	K=K+K3+K4	K=K+K5+K6	K=K+K7+K8		
ADDER			α						
ROM			AL						

Figure 3.17: Data flow of SOR with two RNS multiplier architecture SOR_2M

Table 3.7: Sum of residues reduction algorithm Implementation on Xilinx FPGAs

Architecture	**Platform** FPGA	**Clk frequency** (MHz)	**Latency** (ns)	**Area** (KLUTs),(FFs),(DSPs)	**Throughput** (Mbps)
SOR_1M_N	ARTIX 7	92.5	335	(8.17),(3758),(140)	1576
SOR_1M_N	VIRTEX 7	128.8	241	(8.17),(3758),(140)	2190
SOR_1M_N	KINTEX 7	117.67	263	(8.29),(3758),(140)	2007
SOR_1M_N	VIRTEX US+[1]	192	157	(8.14),(3758),(140)	3363
SOR_1M_N	KINTEX US+	198	156.5	(8.29),(3758),(140)	3373
SOR_1M_P	ARTIX 7	92.5	259.5	(8.73),(4279),(140)	2034
SOR_1M_P	VIRTEX 7	138.8	173	(8.73),(4279),(140)	3052
SOR_1M_P	KINTEX 7	117.6	204	(8.89),(4279),(140)	2588
SOR_1M_P	VIRTEX US+	185.18	130	(8.71),(4279),(140)	4061
SOR_1M_P	KINTEX US+	187.13	128.3	(8.89),(4279),(140)	4115
SOR_2M	ARTIX 7	92.5	194.6	(10.11),(4797),(280)	2713
SOR_2M	VIRTEX 7	128.5	140	(10.11),(4797),(280)	3771
SOR_2M	KINTEX 7	121.9	147.6	(10.27),(4797),(280)	3577
SOR_2M	VIRTEX US+	185.18	97.3	(10.11),(4797),(280)	5426
SOR_2M	KINTEX US+	187.13	96.3	(10.26),(4797),(280)	5482

[1]US+: Ultra Scale+ ™

illustrated on Figure13 of [104], it takes three clock cycles to perform one multiplication and reduction. Therefore, at the maximum working clock frequency, 42 ns will be added to the latency of the proposed RNS modular reduction circuit. As a result, the equivalent latency for an RNS reduction for sQ-RNs and dQ-RNS reduction hardware is 150.53 ns and 168.18 ns, respectively. For the same reason, our proposed MMR algorithm expects two initial modular multiplications which are performed in four clock cycles. This adds 21.2 ns to the circuit's overall latency. On the same FPGA platform used in [104], i.e. KINTEX Ultra Scale+ ™, we achieved the latency of 128.3 ns and 96.3 ns with our SOR_1M_P and SOR_2M designs, respectively. The latency of SOR_2M showed 36% improvement compare to sQ-RNS and 41.1% improvement in contrast to MM_SPA on similar FPGA platforms. Similarly, there is 14.9% and 27.6% improvement of SOR_1M_P latency in compare to sQ-RNS and MM_SPA designs, respectively. The latency of our SOR_M_N, however, is very close to sQ-RNS and MM_SPA designs.

Table 3.8: Comparison of our designs with recent similar works

Design	Platform	Clk frequency (MHz)	Latency (ns)	Area (KLUT),(DSP)	Throughput (Mbps)
[97] MM_PA_P	VIRTEX 6	71.40	14.20	(36.5),(2016) [1]	14798
[97] MM_PA_N	VIRTEX 6	21.16	47.25	(34.34),(2016) [1]	5120
[98] MM_PA_P	VIRTEX 7	62.11	48.3	(29.17),(2799)	15900
[98] MM_SPA	VIRTEX 7	54.34	239.2	(11.43),(512)	1391
[106] SOR_1M_P(Ours)	VIRTEX 7	138.8	173	(8.73),(140)	3052
[106] SOR_2M(Ours)	VIRTEX 7	128.5	140	(10.11),(280)	3771
[104] sQ-RNS	KINTEX US+	139.5	107.53(150.53)[2]	(4.247),(84)	3454 [2]
[104] dQ-RNS	KINTEX US+	142.7	126.14(168.18)[2]	(4.076),(84)	3092 [2]
[106] SOR_1M_P(Ours)	KINTEX US+	187.13	128.3	(8.89),(140)	4115
[106] SOR_2M(Ours)	KINTEX US+	187.13	96.3	(10.26),(280)	5482
MMR(Ours)	KINTEX US+	188.67	116.6 (137.8)[3]	(3.92),(64)	3831

[1]Area reported in [98]

[2]Our estimation for a Modular multiplication.

[3]Modular multiplication latency.

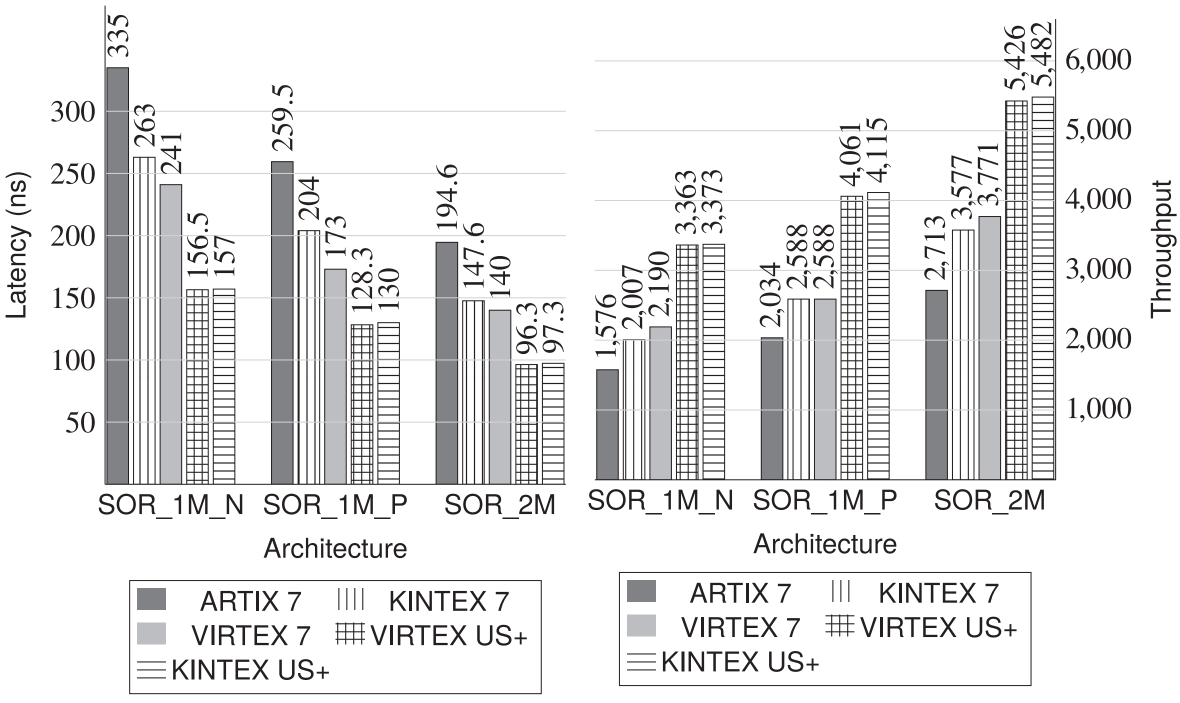


Figure 3.18: SOR architectures Latency and Throughput on Xilinx FPGAs

3.7.4 Hardware test and verification

Numerical example for new RNS Montgomery reduction algorithm

Let us first, bring a numerical example for both RNS Montgomery and SOR reduction algorithms. Considering the moduli set in Table 3.1, The constants for Algorithm 16 are

$$K = \prod_{i=1}^{4} m_i = 29642774844752946015578667808152182207810997947085478414286182832343631559394045.$$

$$Q = \prod_{i=5}^{8} m_i = 29642774844752945679728616558023215758163831990337737268353584222855319684590433.$$

$$K^{-1} = \Big[73642861106762350591, 4995993186629670229, 57357844854190637049, 11577253442093755048\Big].$$

$$Q^{-1} = \Big[37223559108680337996, 108113198197531014, 31185490603629508987, 5269813384330828305\Big].$$

$$R_{K\cup Q} = \Big[274877906944, 19875797281106311556, 24748383423803978097, 4334187607194161013,$$

$$42852602630405522001, 61288154116439491971, 29087400448070938219, 44535440695412410451\Big].$$

$$\Theta_{ij} = \begin{bmatrix} 39190323957512273950 & 32971496243508131580 & 18915118435135040208 & 63680898731287248922 \\ 19595161978756136975 & 15386698246970461404 & 50270465783536849492 & 45822625008708485123 \\ 60239022715817689091 & 3663499582612014620 & 32773420875429090469 & 23921920260709338314 \\ 67012999505327947777 & 655326937912444946 & 21982713152579251378 & 59179623542174329150 \end{bmatrix}.$$

$$\Omega_{ij} = \begin{bmatrix} 10752 & 161280 & 14999040 & 126991872 \\ 12288 & 172032 & 15237120 & 127991808 \\ 14336 & 184320 & 15482880 & 129007616 \\ 21504 & 215040 & 15998976 & 131088384 \end{bmatrix}.$$

$$K_Q = \Big[344064, 10321920, 3839754240, 65019838464\Big],$$

$$-K_Q = \Big[73786976294837862367, 73786976294827884479, 73786976290998451967, 73786976229818367487\Big].$$

Given modulus $p = 2^{256} - 2^{32} - 2^{9} - 2^{8} - 2^{7} - 2^{6} - 2^{4} - 1$, then

$$-p_K = \Big[73498745922981462992, 37758179280169206732, 11240984674211726262, 59663687867699298941\Big].$$

Suppose, $A = 2^{260} - 2^{40} - 123$ and $B = 2^{256} - 135$, are two randomly chosen integers; the product of A and B, i.e. $X = A \times B$ is less than the dynamic range of the RNS base and its presentation in RNS is

$$\Big[6071000727470227668, 24157454150930541429, 38208659493400577367, 26534998718574157803,$$

$$19324504543166081045, 69405015712188571048, 19620781396156619875, 69493056693697113742\Big].$$

The value of $\langle A \cdot B \cdot Q^{-1} \mod p\rangle_{K\cup Q}$ is

$$\Big[25852351324943298246, 8010876674308888376, 53037063643690367473, 61727029647463009885,$$

$$3721892900089140419, 37161346074399323532, 28118412695331475359, 26161054131880081435\Big].$$

From Algorithm 16, $S_{k\cup Q} = \langle A \cdot B\rangle_{k\cup Q} \cdot R_{k\cup Q}$

$$\Big[4561896359882736, 5809795480465295377, 31168081374197409439, 54433645799933486952,$$

$$25116664587954568860, 44369257924514813605, 10844948071510585784, 5339334472289691727\Big].$$

The value of α is calculated by choosing $\Delta = 2^{-4}$ and $q = 4$, that is

$\alpha = \left\lfloor \frac{1}{2^4}\left(\lfloor \frac{25116664587954568860}{2^{62}} \rfloor + \lfloor \frac{44369257924514813605}{2^{62}} \rfloor + \lfloor (\frac{10844948071510585784}{2^{62}} \rfloor + \lfloor \frac{5339334472289691727}{2^{62}} \rfloor + 1\right)\right\rfloor =$ 1.

Then, Z_K is calculated as:

$$Z_K = \left\langle \Theta_{ij} \cdot \begin{bmatrix} 25116664587954568860 \\ 44369257924514813605 \\ 10844948071510585784 \\ 5339334472289691727 \end{bmatrix} + \begin{bmatrix} 73498745922981462992 \\ 37758179280169206732 \\ 11240984674211726262 \\ 59663687867699298941 \end{bmatrix} + \begin{bmatrix} 4561896359882736 \\ 5809795480465295377 \\ 31168081374197409439 \\ 54433645799933486952 \end{bmatrix} \right\rangle_K = \begin{bmatrix} 25852351324943298246 \\ 8010876674308888376 \\ 53037063643690367473 \\ 61727029647463009885 \end{bmatrix}.$$

Now, we calculate t_K matrix.

$$\left\langle \begin{bmatrix} 73642861106762350591 & 0 & 0 & 0 \\ 0 & 49959931866296702229 & 0 & 0 \\ 0 & 0 & 57357844854190637049 & 0 \\ 0 & 0 & 0 & 11577253442093755048 \end{bmatrix} \cdot Z_K \right\rangle_K = \begin{bmatrix} 45201676182137213928 \\ 58456412882758938907 \\ 24868866695340756227 \\ 19111430090298327454 \end{bmatrix} = \begin{bmatrix} t_{k_1} \\ t_{k_2} \\ t_{k_3} \\ t_{k_4} \end{bmatrix}.$$

For the second α at line 4 of Algorithm 16 we choose $\Delta = \frac{3}{2^4}$ and $q = 2^4$. Therefore, α is calculated as

$\alpha = \left\lfloor \frac{1}{2^4}\left(\lfloor \frac{45201676182137213928}{2^{62}} \rfloor + \lfloor \frac{58456412882758938907}{2^{62}} \rfloor + \lfloor (\frac{24868866695340756227}{2^{62}} \rfloor + \lfloor \frac{19111430090298327454}{2^{62}} \rfloor + 3\right)\right\rfloor =$ $\lfloor \frac{33}{2^4} \rfloor = 2$.

Finally, Z_Q is calculated

$$\left\langle \Omega_{ij} \cdot \begin{bmatrix} 45201676182137213928 \\ 58456412882758938907 \\ 24868866695340756227 \\ 19111430090298327454 \end{bmatrix} + \begin{bmatrix} 73786976294837518303 \\ 73786976294817562559 \\ 73786976287158697727 \\ 73786976164798529023 \end{bmatrix} \right\rangle_Q = \begin{bmatrix} 3721892900089140419 \\ 37161346074399323532 \\ 28118412695331475359 \\ 26161054131880081435 \end{bmatrix}.$$

The value of $Z_{K\cup Q}$ which is calculated using Algorithm 16 is equal to the value of $\langle A \cdot B \cdot Q^{-1} \mod p\rangle_{K\cup Q}$ that we directly calculated.

Numerical example for the improved Sum of Residues reduction algorithm

For the same A and B of above example, we calculate modular reduction to p using improved SOM Algorithm 17. The constants used for the sum of residues algorithm are

$M = \prod_{i=1}^{8} m_i = 87869410049671803280001020930911740486246026690322830553846591960069268260578476761508793704538061947379170440345112445787945315766066390580047921338948417148 5.$

$$MI_{\mathcal{B}} = \begin{bmatrix} \langle {M_1}^{-1}\rangle_{m_1} \\ \langle {M_2}^{-1}\rangle_{m_2} \\ \langle {M_3}^{-1}\rangle_{m_3} \\ \langle {M_4}^{-1}\rangle_{m_4} \\ \langle {M_5}^{-1}\rangle_{m_5} \\ \langle {M_6}^{-1}\rangle_{m_6} \\ \langle {M_7}^{-1}\rangle_{m_7} \\ \langle {M_8}^{-1}\rangle_{m_8} \end{bmatrix} = \begin{bmatrix} 73786976294301335551 \\ 44683151288970511775 \\ 61872857499868790663 \\ 63269389302024161492 \\ 13220103208351106415 \\ 19852253190748485956 \\ 8334558314047248845 \\ 10128616081041185010 \end{bmatrix}, \quad PN_{\mathcal{B}} = \begin{bmatrix} \langle -p\rangle_{m_1} \\ \langle -p\rangle_{m_2} \\ \langle -p\rangle_{m_3} \\ \langle -p\rangle_{m_4} \\ \langle -p\rangle_{m_5} \\ \langle -p\rangle_{m_6} \\ \langle -p\rangle_{m_7} \\ \langle -p\rangle_{m_8} \end{bmatrix} = \begin{bmatrix} 73498745922981462992 \\ 37758179280169206732 \\ 11240984674211726262 \\ 59663687867699298941 \\ 45828629812417130916 \\ 18158513701852738400 \\ 73498745922964421837 \\ 73498745922710923722 \end{bmatrix},$$

$$MP_{\mathcal{B}} = \begin{bmatrix} \left\lfloor \frac{\langle M_1 \rangle_p}{2^{184}} \right\rfloor \\ \left\lfloor \frac{\langle M_2 \rangle_p}{2^{184}} \right\rfloor \\ \vdots \\ \left\lfloor \frac{\langle M_8 \rangle_p}{2^{184}} \right\rfloor \end{bmatrix} = \begin{bmatrix} 18027419208564735 \\ 18026098627493887 \\ 18024895592841215 \\ 18022820318855167 \\ 18019845681758207 \\ 18017506445017087 \\ 18015259703164927 \\ 18014838343385087 \end{bmatrix}, \quad MN_{\mathcal{B}} = \begin{bmatrix} \langle \langle -M \rangle_p \rangle_{m_1} \\ \langle \langle -M \rangle_p \rangle_{m_2} \\ \langle \langle -M \rangle_p \rangle_{m_3} \\ \vdots \\ \langle \langle -M \rangle_p \rangle_{m_8} \end{bmatrix} = \begin{bmatrix} 11364192887680349575 \\ 26200618777635377564 \\ 33154480199598314297 \\ 24934496126011739915 \\ 4410582008873787630 \\ 32146213136712920272 \\ 16381009841898477011 \\ 47586101954007074714 \end{bmatrix}.$$

$$H_1 = \begin{bmatrix} \langle \langle M_i \rangle_p \rangle_{m_1} \\ \langle \langle M_i \rangle_p \rangle_{m_2} \\ \vdots \\ \langle \langle M_i \rangle_p \rangle_{m_8} \end{bmatrix} = \begin{bmatrix} 50614143206231599723 & 51631297396430197375 & 52617348558344399635 & 54497831867870420923 \\ 51631297383856305347 & 52632829612579377879 & 53603399600645423851 & 55453342801552763667 \\ 52617348432040390043 & 53603399491679150383 & 54558629083164162115 & 56378313695634768619 \\ 54497830402647991499 & 55453341419316675039 & 56378312477120623987 & 58138324355539210779 \\ 57904135792531963691 & 58800255573705247039 & 59666398330276532179 & 61310442130145772667 \\ 63392687980663313387 & 64176780692571135999 & 64932022254726329491 & 66357641071008049467 \\ 70599434186226143339 & 70900515724409833599 & 71179501484176167187 & 71672874599053936059 \\ 72137976758905910891 & 72299446676089093759 & 72447828010179229459 & 72707013820175051707 \end{bmatrix}, \forall i \in \{1, 2, 3, 4\}$$

$$H_2 = \begin{bmatrix} \langle \langle M_i \rangle_p \rangle_{m_1} \\ \langle \langle M_i \rangle_p \rangle_{m_2} \\ \vdots \\ \langle \langle M_i \rangle_p \rangle_{m_8} \end{bmatrix} = \begin{bmatrix} 57904152923568551179 & 63392856732394127275 & 70612081214764203883 & 72241846743812805739 \\ 58800271729692727651 & 64176936709068135939 & 70911968167315184067 & 72393145718676307651 \\ 59666413476311115835 & 64932165962466763483 & 71189846997373305499 & 72532180550455596955 \\ 61310454639453388651 & 66357761122263751179 & 71681262439500872139 & 72775040254836920011 \\ 64257388400519588299 & 68881319813637788779 & 72418499498230600747 & 73130997117418219819 \\ 68881245460061363851 & 72712513172963830059 & 73004798685022841067 & 73412478982076841451 \\ 72413049724370082571 & 73001695670686426539 & 62915464440731821419 & 73683476982486055531 \\ 73086900245091746059 & 73387310215928105899 & 73676957585362062187 & 56771868068651819628 \end{bmatrix}, \forall i \in \{5, 6, 7, 8\}$$

The matrix H is obtained by concatenation of matrix H_2 to the right side of matrix H_1. That is, $H = \left[H_1 || H_2 \right]$.

The RNS form of $X = A \cdot B$ over base $\mathcal{B}$ was calculated in the last example. The value of γ_i matrix is calculated as

$$\gamma_{\mathcal{B}} = \left\langle \begin{bmatrix} 6071000727470227668 \times 7378697629430133551 \\ 24157454150930541429 \times 44683151288970511775 \\ 38208659493400577367 \times 61872857499868790663 \\ 26534998718574157803 \times 63269389302024161492 \\ 19324504543166081045 \times 13220103208351106415 \\ 69405015712188571048 \times 19852253190748485956 \\ 1962078139615661 9875 \times 8334558314047248845 \\ 69493056693697113742 \times 10128616081041185010 \end{bmatrix} \right\rangle_{\mathcal{B}} = \begin{bmatrix} 2305834099259866056 \\ 60366484759219081820 \\ 43977497233343938196 \\ 35776005214296587945 \\ 14400836904693571174 \\ 15090841826745158153 \\ 25687384790642700856 \\ 23774058454823196884 \end{bmatrix}.$$

Taking $q = 8$ and $\Delta = 2^{-4}$ for Algorithm 17, we can calculate the value of α as follows

$$\alpha = \left\lfloor \frac{1}{2^8} \left(\left\lfloor \frac{2305834099259866056}{2^{58}} \right\rfloor + \left\lfloor \frac{60366484759219081820}{2^{58}} \right\rfloor + \ldots + \left\lfloor \frac{23774058454823196884}{2^{58}} \right\rfloor + 2^4 \right) \right\rfloor = \left\lfloor \frac{780}{256} \right\rfloor = 3.$$

Then, the value of $AM_{\mathcal{B}} = \langle -\alpha M \rangle_{\mathcal{B}}$ is

$$AM_{\mathcal{B}} = \left\langle 3 \times \begin{bmatrix} 11364192887680349575 \\ 26200618777635377564 \\ 33154480199598314297 \\ 24934496126011739915 \\ 4410582008873787630 \\ 32146213136712920272 \\ 16381009841898477011 \\ 47586101954007074714 \end{bmatrix} \right\rangle_{\mathcal{B}} = \begin{bmatrix} 34092578663041048725 \\ 4814880038067926233 \\ 25676464303956736436 \\ 1016512083197013298 \\ 13231746026621362890 \\ 22651663115300554417 \\ 49143029525695431033 \\ 68971329567183018191 \end{bmatrix}.$$

The value of κ is calculated as

$$\kappa = \left\lfloor \frac{1}{2^{72}} \Big(2305834099259866056 \times 18027419208564735 + 6036648475921908 1820 \times 18026098627493887 + \ldots + 2377405845482319 6884 \times 18014838343385087 \Big) \right\rfloor = 844844570355236.$$

Then the values of $\langle \kappa \cdot -p \rangle_{\mathcal{B}}$ can be calculated.

$$KP_{\mathcal{B}} = \left\langle 844844570355236 \times \begin{bmatrix} 7349874592298146 2992 \\ 3775817928016920 6732 \\ 1124098467421172 6262 \\ 5966368786769929 8941 \\ 4582862981241713 0916 \\ 1815851370185273 8400 \\ 7349874592296442 1837 \\ 7349874592271092 3722 \end{bmatrix} \right\rangle_{\mathcal{B}} = \begin{bmatrix} 2190272024968267 7206 \\ 6340583510688912 4156 \\ 6799793129184159 6727 \\ 4007383378231461 9891 \\ 5488246592493272 3288 \\ 3677952949822377 9834 \\ 1345125877660432 4263 \\ 5075282502016294 4228 \end{bmatrix}$$

The matrix Y is then calculated by $Y = \langle H \cdot \gamma_{\mathcal{B}} \rangle_{\mathcal{B}}$ which results

$$Y_{\mathcal{B}} = \langle H \cdot \gamma_{\mathcal{B}} \rangle_{\mathcal{B}} = \begin{bmatrix} 1807910643764090 0505 \\ 4159425684821983 2059 \\ 4265777100333602 9507 \\ 4681911754013515 6306 \\ 3363030950937487 0350 \\ 6998344495796901 2809 \\ 1148011704808189 6423 \\ 2813722705812737 7494 \end{bmatrix}$$

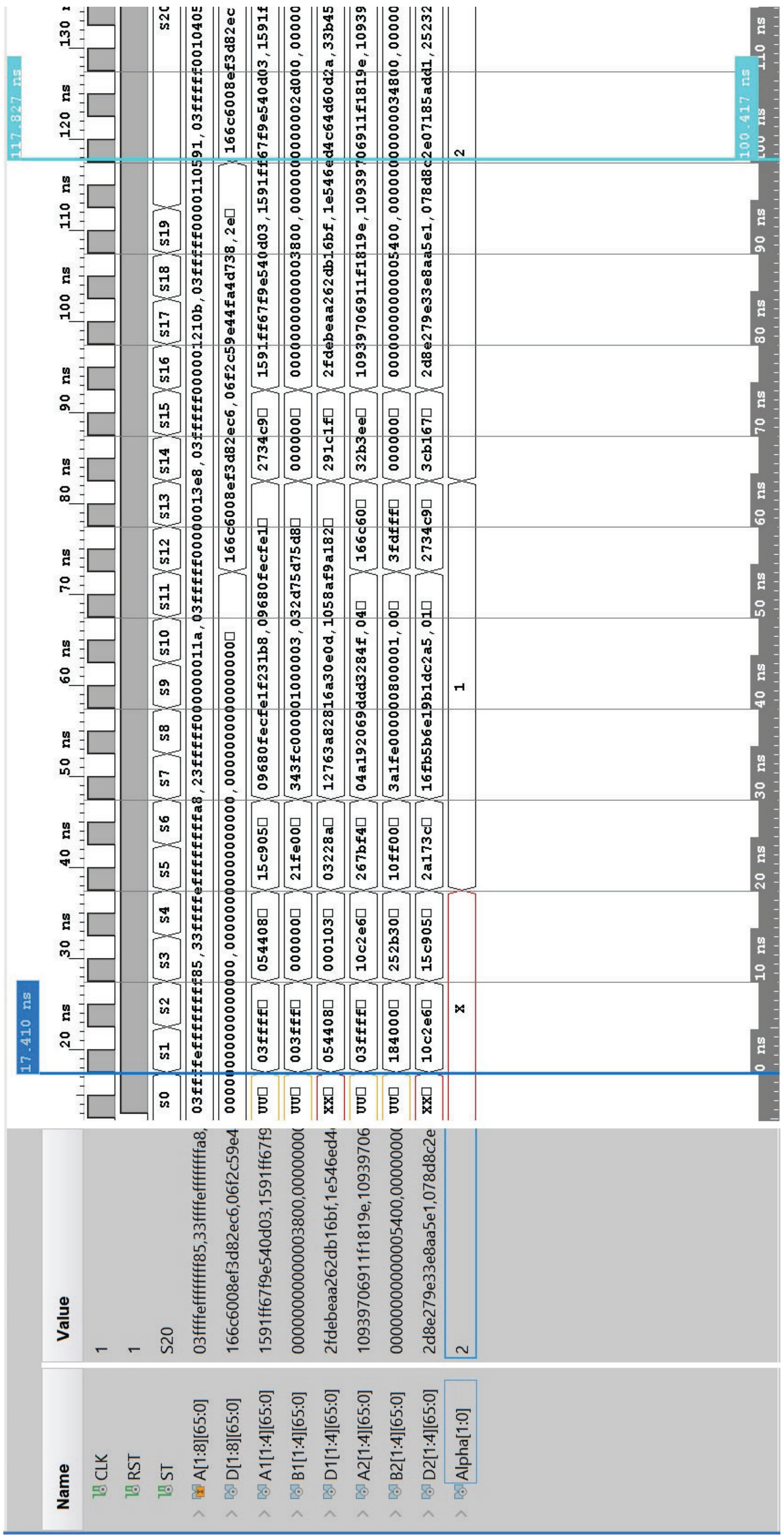
Finally, the value of $Z = X \mod p$ is obtained by adding $Y_{\mathcal{B}}$, $AM_{\mathcal{B}}$, and $KP_{\mathcal{B}}$ which is

$$Z = \left\langle \begin{bmatrix} 1807910643764090 0505 \\ 4159425684821983 2059 \\ 4265777100333602 9507 \\ 4681911754013515 6306 \\ 3363030950937487 0350 \\ 6998344495796901 2809 \\ 1148011704808189 6423 \\ 2813722705812737 7494 \end{bmatrix} + \begin{bmatrix} 3409257866304104 8725 \\ 481488003806792 6233 \\ 2567646430395673 6436 \\ 101651208319701 3298 \\ 1323174602662136 2890 \\ 2265166311530055 4417 \\ 4914302952569543 1033 \\ 6897132956718301 8191 \end{bmatrix} + \begin{bmatrix} 2190272024968267 7206 \\ 6340583510688912 4156 \\ 6799793129184159 6727 \\ 4007383378231461 9891 \\ 5488246592493272 3288 \\ 3677952949822377 9834 \\ 1345125877660432 4263 \\ 5075282502016294 4228 \end{bmatrix} \right\rangle_{\mathcal{B}} = \begin{bmatrix} 287429055526419973 \\ 3602799569833867 5989 \\ 6254519030429615 6215 \\ 1412248711080858 3048 \\ 2795754516609075 0097 \\ 5562766127665514 0661 \\ 287429055543445512 \\ 287429055796928011 \end{bmatrix}$$

Integer value of the modular multiplication $C = A \cdot B \mod p =$ 115792089237316195423570985008687907853269984665640564035030364628902211960385, which its representation in RNS base $\mathcal{B}$ is identical to the values obtained from the SOM Algorithm 17.

Hardware verification and testbench

To automate hardware verification, using a Python program that can be found in SectionB.5.1, we generated random 256-bit integers, their RNS, and the RNS Modulo p. The results are saved in a text file which is used by the VHDL testbench (listed in B.5.2). The VHDL program feeds the inputs with all random RNS numbers and compares the hardware output with the RNS number modulo p. Figure 3.19 and Figure 3.20 show the hardware test results on the new MMR and the improved SOR reduction hardware implementation on a KINTEX UltaScale + FPGA. The proposed algorithms 16 and 17 were validated by Maple simulation. The Maple codes for MMR and SOM algorithms validation are presented in Appendix A.1 and A.2, respectively.



Name	Value
CLK	1
RST	1
ST	S20
A[1:8][65:0]	03fffefffffff85,33ffffeffffffffa8,
D[1:8][65:0]	166c6008ef3d82ec6,06f2c59e4
A1[1:4][65:0]	1591ff67f9e540d03,1591ff67f9
B1[1:4][65:0]	00000000000003800,0000000
D1[1:4][65:0]	2fdebeaa262db16bf,1e546ed4
A2[1:4][65:0]	10939706911f1819e,10939706
B2[1:4][65:0]	00000000000005400,0000000
D2[1:4][65:0]	2d8e279e33e8aa5e1,078d8c2e
Alpha[1:0]	2

Figure 3.19: Hardware simulation of the proposed RNS Montgomery reduction algorithm.

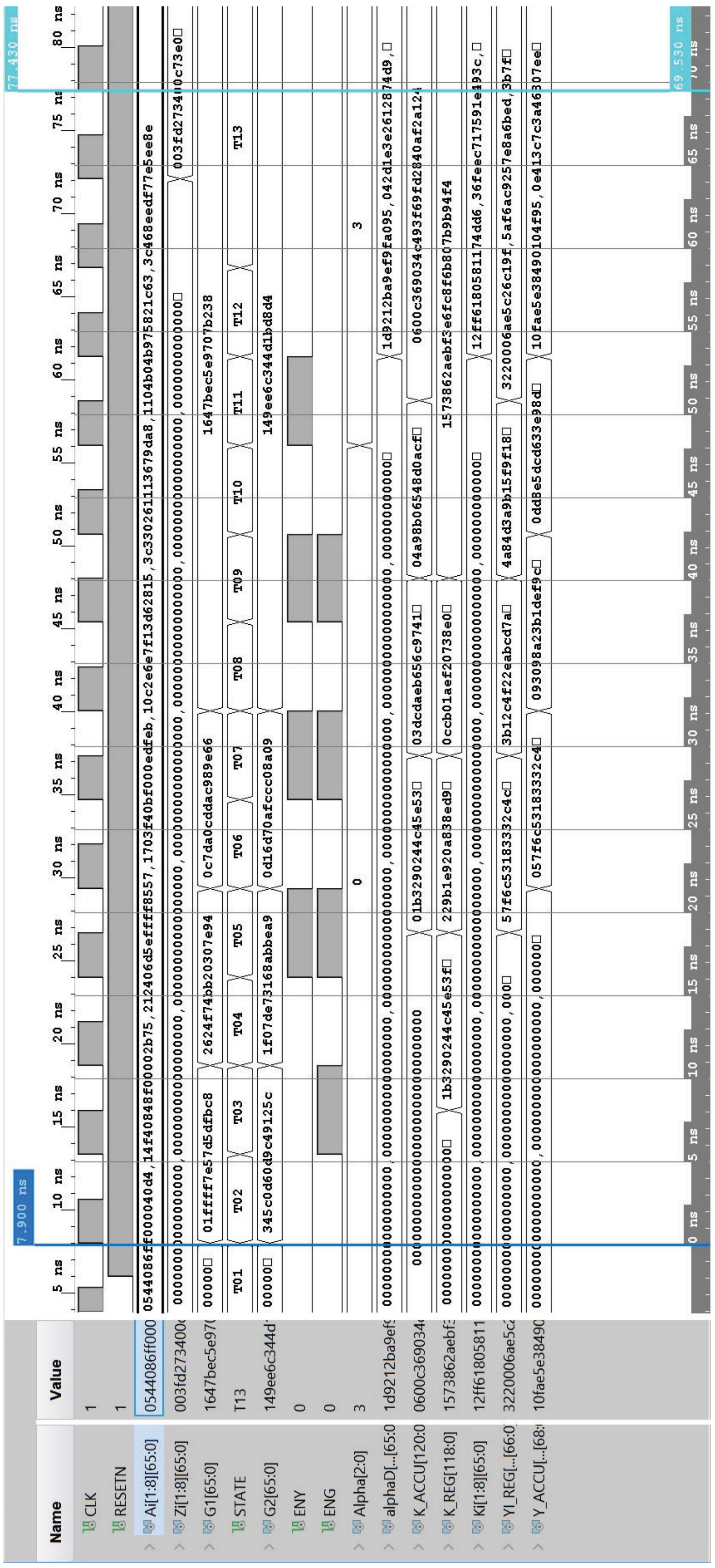

Name
Value
CLK 1
RESETN 1
Ai[1:8][65:0] 0544086ff000
Zi[1:8][65:0] 003fd273400
G1[65:0] 1647bec5e970
STATE T13
G2[65:0] 149ee6c344d
ENY 0
ENG 0
Alpha[2:0] 3
alphaD[...[65:0 1d9212ba9ef
K_ACCU[120:0 0600c369034
K_REG[118:0] 1573862aebf3
Ki[1:8][65:0] 12ff61805811
YI_REG[...[66:0 3220006ae5c
Y_ACCU[...[68: 10fae5e38490
7.900 ns
77.430 ns
69.530 ns

Figure 3.20: Hardware simulation of the proposed sum of residues reduction algorithm.

4 RNS Elliptic curve point multiplication Hardware design

4.1 Introduction

In this chapter, we first review the ECC arithmetic in various coordinate systems in the context of RNS and propose hardware architecture for each coordinate system based on the primitives introduced in Chapter 3. Then we implement the point multiplication core by applying Double-and-Add, NAF, and DBC algorithms on SECP256K1, ED25519, and Brainpool256r1 curves. For the Curve SECP256K1 which has an efficient endomorphism, we propose a new low-latency low-cost point multiplication hardware using GLV method. Comparing our design with the latest similar works in the literature, we show the impacts of using our new arithmetic algorithms as well as applying efficient scalar multiplication algorithms on the latency and performance of the ECC point multiplication co-processor.

4.2 New RNS ECC Point arithmetic Design[1]

In the RNS context, unlike integer arithmetic, the bit-length of channels does not increase by performing addition and multiplication operations. Accordingly, it is permissible to perform as many operations as we can within the dynamic range before we reduce the result modulo p. From chapter 3, our designed RNS has a 528-bit dynamic range. Therefore, until the result is not exceeding 528-bit long, we can perform arithmetic operations without requiring a modular reduction which is costly in terms of operation latency. For instance, it is possible to perform a 256×256-bit multiplication, add the result to a 512-bit number, multiply the outcome by a 2^{15}-bit number and finally perform a modular reduction. This property of RNS operations leads to achieving more efficient ECC point arithmetic than the traditional integer arithmetic. In RNS arithmetic, we can do one multiplication and several additions or subtractions before the result exceeds the dynamic range. A modular reduction is performed afterwards. In integer

[1]This section is a part of the published work [109]

arithmetic, however, a modular reduction is necessary after each multiplication.

In the following, we propose parallel RNS point arithmetic for short Weierstraß curves SECP256K1, Brainpool256r1, and twisted Edwards curve ED25519. The efficient explicit formulae for point doubling, point addition, and point tripling are presented in EFD database [26]. Based on these formulae and combining properties of parallel processing and RNS arithmetic, we designed new point arithmetic formulae that have a smaller number of modular reductions and process in parallel, hence, notably improve the speed of calculations. In our design, two modular reduction units are used which can work in parallel. We classified the operations that can be performed in parallel or during a modular reduction in a "*logic level*". Then we rearranged operations — when possible — to reduce the number of logic levels. A Squaring operation is more efficient than a multiplication in terms of speed. Nevertheless, it needs the implementation of new hardware. To reduce the logic area, we used the same hardware for RNS multiplication and squaring. In the flow diagrams, however, these operations are displayed differently.

4.2.1 Point arithmetic on Koblitz Curve SECP256K1

Point doubling

Take P_1 a point on the curve SECP256K1, represented in Jacobian coordinates (X_1, Y_1, Z_1), all in RNS form. The coordinates of point $P_2 = 2P_1$, that is, (X_2, Y_2, Z_2) are calculated using formulas in (4.1). We derived these formulas by applying RNS and parallel processing properties on efficient point-doubling formulas (6) proposed by Cohen et al. in [110].

$$\begin{aligned}
(A, B) &\leftarrow (\langle 3{X_1}^2\rangle_p, \langle 2{Y_1}^2\rangle_p) \\
C &\leftarrow 2X_1B \\
(X_2, Z_2) &\leftarrow (\langle A^2 - 2C\rangle_p, \langle 2Y_1Z_1\rangle_p) \\
D &\leftarrow \langle C - X_2\rangle_p \\
Y_2 &\leftarrow \langle AD - 2B^2\rangle_p.
\end{aligned} \tag{4.1}$$

The operations inside the parenthesis are performed in parallel. The arrow sign is used to indicate updating the intermediate registers or outputs on the edge of the clock in hardware. The detailed data flow is depicted in Figure 4.1a. A modular reduction is not required to calculate C, A^2, B^2, and AD as their values are less than the dynamic range and the next operation is a modular subtraction which is always followed by a modular reduction. The total number of RNS modular reductions is six compared to the seven reductions required in integer arithmetic [110].

Point addition

Take P_1 and P_2 two points on the curve SECP256K1, represented in Jacobian coordinates by (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2), respectively. The point $P_3(X_3, Y_3, Z_3)$ is a point on a curve given by $P_3 = P_1 + P_2$, which its coordinates can be calculated using formulas in (4.2). The RNS parallel formulae are directly derived from formulas (5) in [110] by application of RNS and parallel processing properties.

$$\begin{aligned}
(A_1, A_2) &\leftarrow (\langle {Z_1}^2\rangle_p, \langle {Z_2}^2\rangle_p) \\
(A_3, A_4) &\leftarrow (\langle A_1 Z_1\rangle_p, \langle A_2 Z_2\rangle_p) \\
(B_1, H_1) &\leftarrow (\langle Y_1 A_4\rangle_p, \langle Z_1 Z_2\rangle_p) \\
(B_2, C_2) &\leftarrow (Y_2 A_3, X_2 A_1) \\
(C_1, D_1) &\leftarrow (\langle X_1 A_2\rangle_p, \langle B_2 - B_1\rangle_p) \\
(D_2, E_1) &\leftarrow (\langle C_2 - C_1\rangle_p, {D_1}^2) \\
(Z_3, E_2) &\leftarrow (\langle H_1 D_2\rangle_p, \langle {D_2}^2\rangle_p) \\
(F_1, F_2) &\leftarrow (C_1 E_2, \langle D_2 E_2\rangle_p) \\
G_1 &\leftarrow F_2 + 2F_1 \\
X_3 &\leftarrow \langle E_1 - G_1\rangle_p \\
G_2 &\leftarrow \langle F_1 - X_3\rangle_p \\
Y_3 &\leftarrow \langle D_1 G_2 - B_1 F_2\rangle_p.
\end{aligned} \tag{4.2}$$

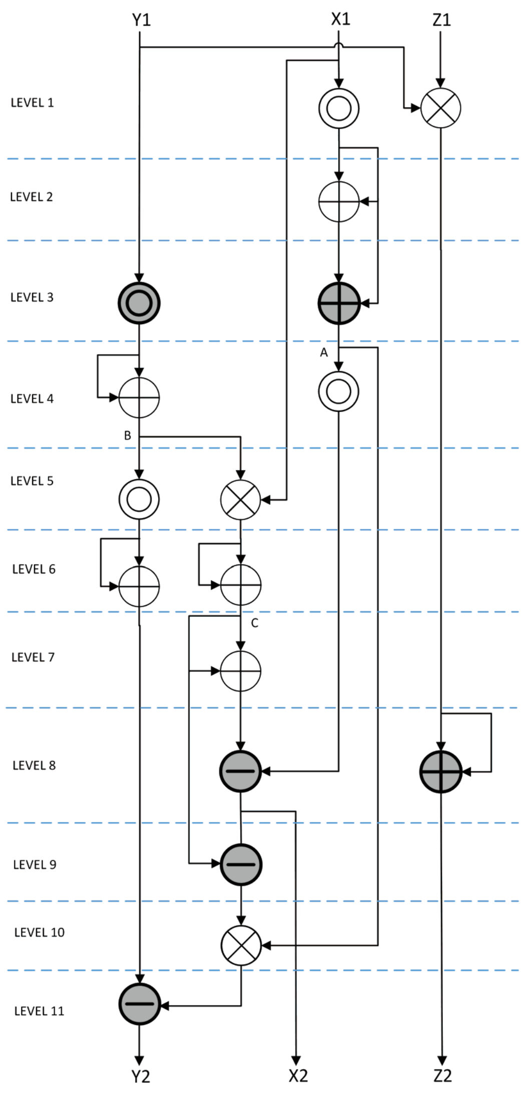
Here, 15 modular reductions are required. That is one less than the modular reduction counts in integer arithmetic [26]. Using two parallel SOR, the implementation can be done in 12 logic levels, as presented in Figure 4.1b.

Point tripling

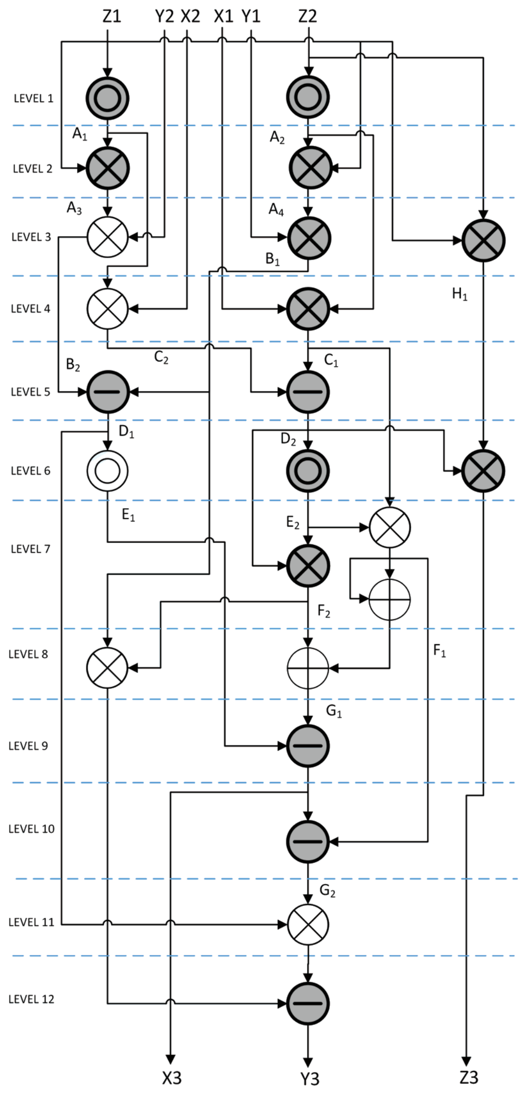
Point tripling (ECPT) is used to implement scalar multiplications based on the DBC algorithms discussed in chapter 2. Given point P_1 on the curve SECP256K1 with Jacobian coordinates (X_1, Y_1, Z_1), all in RNS form, a point tripling is an operation that returns the point $P_3 = 3P_1$ on the curve whose coordinates (X_3, Y_3, Z_3) are obtained from formulas in (4.3). Efficient point tripling formulas for short Weirstraß curves were proposed by Dimitrov et al. in [50]. The RNS point tripling formulas (4.3) are resulted from applying RNS and parallel processing properties on Dimitrov formulas.

Point tripling calculation in RNS requires ten modular reductions. When compared to the fifteen modular reductions required for ECPT calculation in integer arithmetic [26], the RNS improves the ECPT latency efficiently, which makes the application of RNS in the design of a low-latency ECC co-processor more appealing. As shown in Figure 4.2, by utilising two modular reduction units, the ECPT hardware implementation can be realised in 17 logic levels in which six logic levels include modular reductions.

$$\begin{aligned}
(A, B) &\leftarrow (\langle 3{X_1}^2\rangle_p, \langle 2{Y_1}^2\rangle_p) \\
C_1 &\leftarrow (\langle 12 X_1 B - A^2\rangle_p \\
(Z_3, C_2) &\leftarrow (\langle Z_1 C_1\rangle_p, \langle {C_1}^2\rangle_p) \\
(D, C_3) &\leftarrow (4B^2, C_1 C_2) \\
(E_1, E_2) &\leftarrow (\langle D - AC_1\rangle_p, \langle 2D - AC_1\rangle_p) \\
(F, X_3) &\leftarrow (\langle (4E_2E_1 + C_3)\rangle_p, \langle 4BE_1 + X_1C_2\rangle_p) \\
Y_3 &\leftarrow \langle p - Y_1 F\rangle_p.
\end{aligned} \tag{4.3}$$



(a) ECPD on SEC2P256K1



(b) ECPA on SEC2P256K1

Figure 4.1: RNS point doubling and point addition flow diagram on SECP256k1

4.2.2 Point arithmetic on twisted Edwards curve ED25519

Point doubling

Point doubling formulas on twisted Edwards curve were introduced by Bernstein et al. in [111]. A point doubling in integer arithmetic can be performed by eight modular reductions. The parallel RNS point doubling, however, requires seven modular reduction operations. The hardware implementation is achievable in eight logic levels using two parallel modular reduction units. Take P_1 a point on the curve ED25519 represented in projective coordinates (X_1, Y_1, Z_1), all in RNS form, the coordinates of the point $P_2 = 2P_1$ are calculated using formulas in (4.4). Figure 4.3a shows the data flow diagram of RNS point doubling on the twisted-Edwards curve

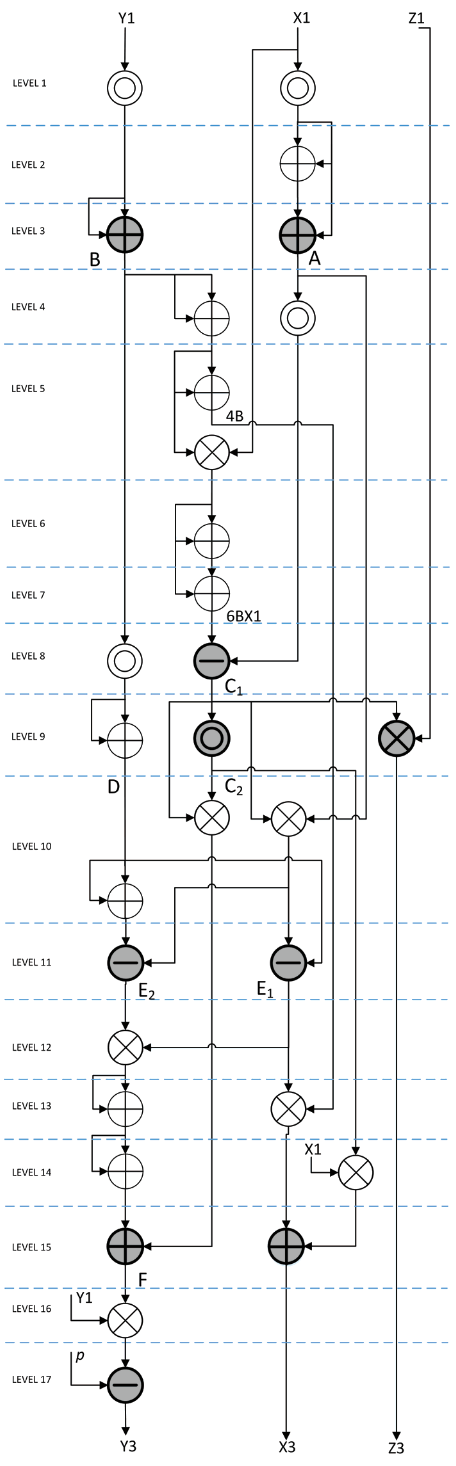


Figure 4.2: ECPT flow diagram on SECP256K1

ED25519.

$$\begin{aligned}
(A, B) &\leftarrow (2{Z_1}^2, (X_1 + Y_1)^2) \\
(C, D) &\leftarrow ({X_1}^2, {Y_1}^2) \\
E &\leftarrow \langle C + D\rangle_p \\
(J, F) &\leftarrow (\langle B - E\rangle_p, \langle C - D\rangle_p) \\
H_1 &\leftarrow F + A \\
(Y_2, H_2) &\leftarrow (\langle EF\rangle_p, \langle p - H_1\rangle_p) \\
(X_2, Z_2) &\leftarrow (\langle H_2 J\rangle_p, \langle 2H_2 F\rangle_p).
\end{aligned} \tag{4.4}$$

Point addition

Take P_1 and P_2 the two points on the curve ED25519 represented in projective coordinates (X_1, Y_1, Z_1) and $P_2(X_2, Y_2, Z_2)$ in RNS form. The coordinates of the third point P_3 on the curve defined as the addition of points P_1 and P_2 i.e. $P_3 = P_1 + P_2$, can be obtained from formulas in (4.5). These formulas are directly derived from Bernstein [111] point addition formulas in integer arithmetic by applying RNS and parallel processing properties. Point addition on twisted Edwards curves in integer arithmetic can be performed by thirteen modular reductions.

$$\begin{aligned}
(A, B) &\leftarrow (\langle X_1X_2\rangle_p, \langle Y_1Y_2\rangle_p)\\
(C_1, C_2) &\leftarrow (X_1 + Y_1, X_2 + Y_2)\\
(D_1, D_2) &\leftarrow (AB, A + B)\\
(E_1, E_2) &\leftarrow (\langle C_1C_2 - D_2\rangle_p, \langle Z_1Z_2\rangle_p)\\
E_3 &\leftarrow {E_2}^2\\
(F, G) &\leftarrow (\langle E_3 + D_1\rangle_p, \langle E_3 - D_1\rangle_p)\\
(I_1, I_2) &\leftarrow (\langle E_2F\rangle_p, \langle E_2G\rangle_p)\\
(X_3, Y_3) &\leftarrow (\langle E_1I_1\rangle_p, \langle D_2I_2\rangle_p)\\
Z_3 &\leftarrow \langle FG\rangle_p.
\end{aligned} \tag{4.5}$$

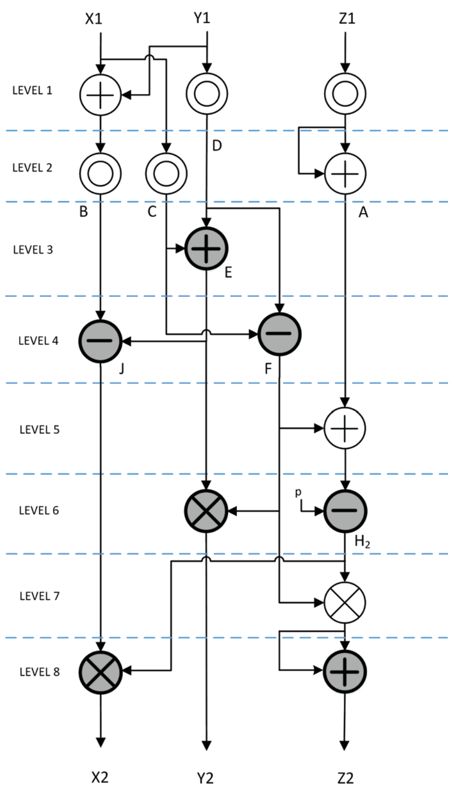
In comparison, our proposed RNS point-addition on the twisted Edwards curve requires eleven modular reduction operations. The hardware implementation can be performed in eight logic levels when using two modular reduction units in parallel. Figure 4.3b shows the data flow of the RNS point addition on the curve ED25519.

Point tripling

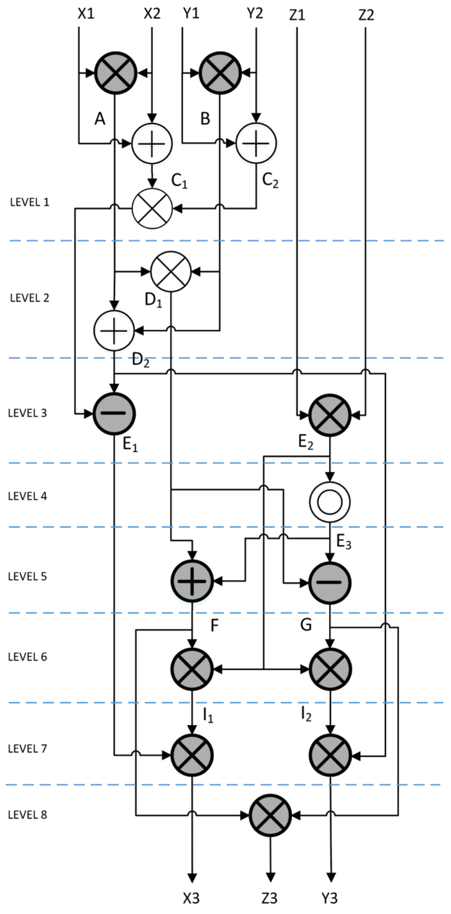
The RNS projective coordinates of point $P_3 = 3P_1$ on the twisted Edwards curve can be obtained from formulas in (4.6), which are concluded from Bernstein point tripling formulas in integer arithmetic [111] after applying RNS and parallel processing properties.

$$\begin{aligned}
(A, B) &\leftarrow (\langle {Y_1}^2\rangle_p, \langle p - {X_1}^2\rangle_p)\\
C &\leftarrow A + B\\
(D_1, D_2) &\leftarrow (2\langle 2{Z_1}^2 - C\rangle_p, \langle A - B\rangle_p)\\
(E_1, E_2) &\leftarrow (D_1B, D_1A)\\
F &\leftarrow CD_2\\
(G_1, G_2) &\leftarrow (\langle F - E_2\rangle_p, \langle F + E_1\rangle_p)\\
(H_1, H_2) &\leftarrow (\langle E_2 + F\rangle_p, \langle E_1 - F\rangle_p)\\
(I_1, I_2) &\leftarrow (\langle X_1G_1\rangle_p, \langle Y_1G_2\rangle_p)\\
(I_3, X_3) &\leftarrow (\langle Z_1G_1\rangle_p, \langle H_1I_1\rangle_p)\\
(Y_3, Z_3) &\leftarrow (\langle H_2I_2\rangle_p, \langle G_2I_3\rangle_p).
\end{aligned} \tag{4.6}$$

RNS point tripling on a twisted Edwards curve needs 14 field reductions and can be implemented with 11 logic levels using two SOR reduction units. Figure 4.4 depicts the data flow diagram in hardware. Point doubling and point addition on twisted Edwards curves use a smaller number of modular reductions than the short Weierstraß curves like SECP256K1 and Brainpool256r1. However, point tripling on twisted Edwards curve requires more modular reduction numbers.



(a) ECPD on ED25519



(b) ECPA on ED25519

Figure 4.3: RNS point doubling and point addition flow diagram on ED25519

4.2.3 Point arithmetic on Brainpool256r1

Brainpool256r1 is a short Weierstraß curve. The point addition on this curve is the same as on the curve SECP256K1. However, in point doubling and point tripling formulas, the non-zero domain parameter a must be taken into account. The Brainpool256r1 field characteristic, p is not in the form of pseudo-Mersenne. Consequently, the condition (3.37) is not valid for Brainpool256r1, and the SOR reduction algorithm does not apply to this curve. Accordingly, we used our proposed RNS Montgomery modular reduction implementation detailed in section 3.6, to implement point arithmetic units on this curve. Bearing in mind that the output of Montgomery reduction is a multiple of Q^{-1}, a correction factor has to be taken into account. We assume that the input of the reduction unit is a multiple of Q, in other words, we use the mapping $(X, Y, Z) \to (\langle XQ\rangle_p, \langle YQ\rangle_p, \langle ZQ\rangle_p)$. Then, rearrange the operations such that the output of the ECPD, ECPA, and ECPT units will be a multiple of Q as well. The point arithmetic on the Brainpool256r1 curve, using Montgomery reduction are then introduced as follows.

Point doubling

Take the point $P_1(X_1, Y_1, Z_1)$ in Jacobian coordinates on the Brainpool256r1 curve, the coordinates of the point $P_2 = 2P_1$ on the curve are calculated using formulas in (4.7). If the coordinates of the point P_1 are mapped to $(\langle X_1Q\rangle_p, \langle Y_1Q\rangle_p, \langle Z_1Q\rangle_p)$ then P_2 coordinates will

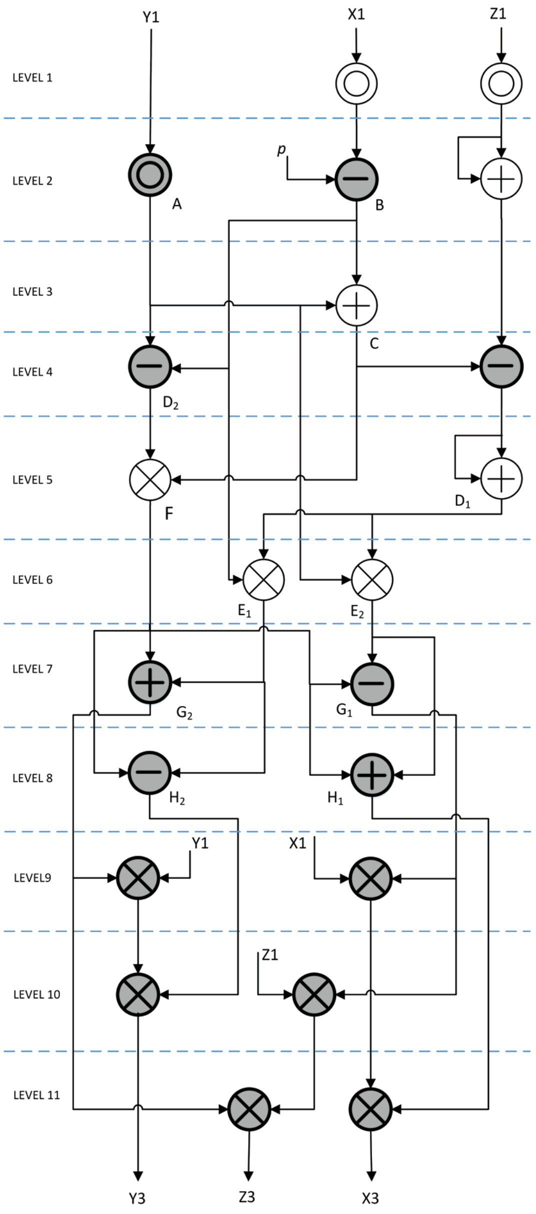


Figure 4.4: ECPT flow diagram on ED25519

have a factor of Q as well. For ease of writing, we did not show the factor Q in the formulas. Brainpool256r1 point doubling requires 9 modular reductions in 9 logic levels. Figure 4.5a depicts the point doubling logic levels on this curve.

$$\begin{aligned}
(Z', B) &\leftarrow (\langle {Z_1}^2\rangle_p, 2\langle {Y_1}^2\rangle_p) \\
(Z, A_1) &\leftarrow (\langle {Z'}^2\rangle_p, 3{X_1}^2) \\
(A, C) &\leftarrow (\langle A_1 + a \cdot Z\rangle_p, 2X_1 B) \\
(X_2, Z_2) &\leftarrow (\langle A^2 - 2C\rangle_p, \langle 2Y_1 Z_1\rangle_p) \\
D &\leftarrow \langle C - X_2\rangle_p \\
Y_2 &\leftarrow \langle A \cdot D - 2B^2\rangle_p
\end{aligned} \tag{4.7}$$

Point addition

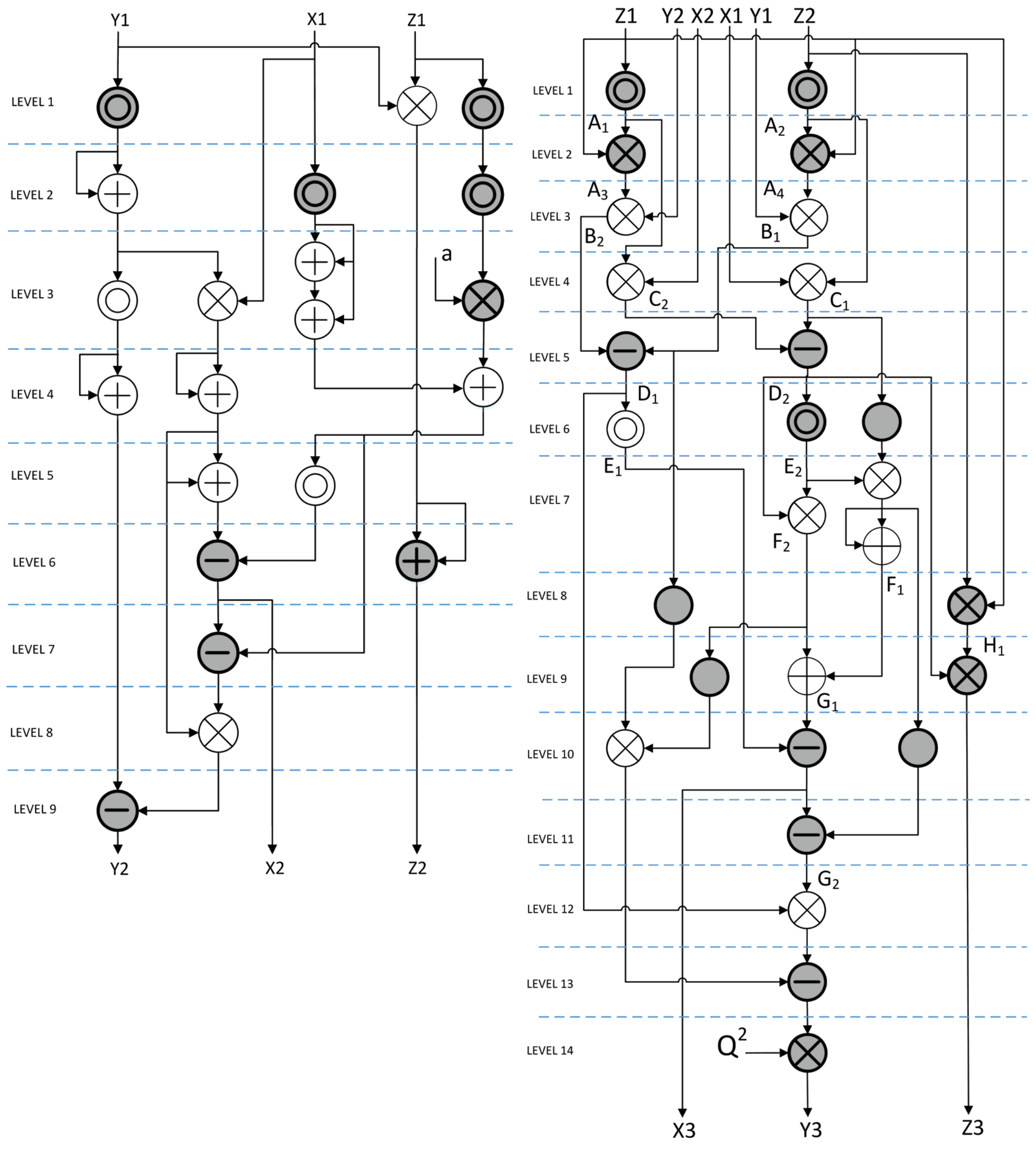
Our Brainpool256r1 hardware is using the RNS Montgomery reduction unit, which adds the factor Q^{-1} to the output, we need to rearrange arithmetic operations to manage this factor. It is assumed that the points $P_1(X_1, Y_1, Z_1)$ and $P_2(X_2, Y_2, Z_2)$ on the curve Brainpool256r1 with Jacobian coordinates, are mapped to $(\langle X_1Q\rangle_p, \langle Y_1Q\rangle_p, \langle Z_1Q\rangle_p)$ and $(\langle X_2Q\rangle_p, \langle Y_2Q\rangle_p, \langle Z_2Q\rangle_p)$, respectively. Then, the mapped coordinates of the point $P_3(X_3, Y_3, Z_3) = P_1 + P_2$ on the curve – i.e. $(\langle X_3Q\rangle_p, \langle Y_3Q\rangle_p, \langle Z_3Q\rangle_p)$ – are obtained using formulas (4.8). For ease of presentation, the factor Q is not shown.

$$\begin{aligned}
(A_1, A_2) &\leftarrow (\langle {Z_1}^2\rangle_p, \langle {Z_2}^2\rangle_p)\\
(A_3, A_4) &\leftarrow (\langle A_1Z_1\rangle_p, \langle A_2Z_2\rangle_p)\\
(B_1, B_2) &\leftarrow (Y_1A_4, X_2A_1)\\
(C_1, C_2) &\leftarrow (X_1A_2, X_2A_1)\\
(D_1, D_2) &\leftarrow (\langle B_2 - B_1\rangle_p, \langle C_2 - C_1\rangle_p)\\
(E_2, F_1) &\leftarrow (\langle {D_2}^2\rangle_p, \langle C_1\rangle_p E_2)\\
(C_2, H_1) &\leftarrow (\langle C_2\rangle_p, \langle Z_1Z_2\rangle_p)\\
F_2 &\leftarrow D_2E_2\\
(X_3, F_1) &\leftarrow (\langle E_1 - (F_2 + 2F_1)\rangle_p, \langle F_1\rangle_p).\\
(Z_3, F_2) &\leftarrow (\langle D_2H_1\rangle_p, \langle F_2\rangle_p)\\
G_2 &\leftarrow \langle F_1 - X_3\rangle_p\\
Y_3 &\leftarrow \langle D_1G_2 - B_1F_2\rangle_p\\
Y_3 &\leftarrow \langle Y_3Q^2\rangle_p.
\end{aligned} \tag{4.8}$$

Figure 4.5b shows the data flow diagram of point addition on the Brainpool256r1 curve. Point addition is performed using 17 modular reductions in 14 logic levels.

Point tripling

Given the point $P_1(X_1, Y_1, Z_1)$ with Jacobian coordinates on Brainpool256r1 curve, point tripling is an operation that yields the point $P_3(X_3, Y_3, Z_3)$ on the curve denoted by $P_3 = 3P_1$. The point P_3 coordinates are obtained from formulas in (4.9). RNS Montgomery reduction unit is used in our point tripling hardware design. Same as ECPD and ECPA, we assume that the input coordinates (X_1, Y_1, Z_1) are mapped to $(\langle X_1Q\rangle_p, \langle Y_1Q\rangle_p, \langle Z_1Q\rangle_p)$. In this case, the output of (4.9) are multiples of Q, i.e. $(\langle X_3Q\rangle_p, \langle Y_3Q\rangle_p, \langle Z_3Q\rangle_p)$. For a better readability, we did not



(a) ECPD on Brainpool256r1

(b) ECPA on Brainpool256r1

Figure 4.5: RNS point doubling and point addition flow diagram on Brainpool256r1

show the coefficient Q in the formulas (4.9).

$$
\begin{aligned}
(Z', A_1) &\leftarrow (\langle {Z_1}^2 \rangle_p, 3{X_1}^2) \\
(Z, B) &\leftarrow (\langle Z'^2 \rangle_p, \langle 2{Y_1}^2 \rangle_p) \\
A_2 &\leftarrow a \cdot Z \\
A &\leftarrow \langle A_1 + A_2 \rangle_p \\
(C_1, C_2) &\leftarrow (\langle 6X_1B - A^2 \rangle_p, \langle {C_1}^2 \rangle_p) \\
(D, C_3) &\leftarrow (2B^2, C_1C_2) \\
(E_1, E_2) &\leftarrow (\langle D - AC_1 \rangle_p, \langle 2D - AC_1 \rangle_p) \\
(F, X_3) &\leftarrow (\langle p - (4E_2E_1 + C_3) \rangle_p, \langle 4BE_1 + X_1C_2 \rangle_p) \\
(Y_3, Z_3) &\leftarrow (\langle Y_1F \rangle_p, \langle Z_1C_1 \rangle_p)
\end{aligned}
\tag{4.9}
$$

The RNS Point tripling requires 12 field reductions on the Brainpool256r1 curve. Figure 4.6 illustrates logic levels of this operation using two parallel modular reduction units.

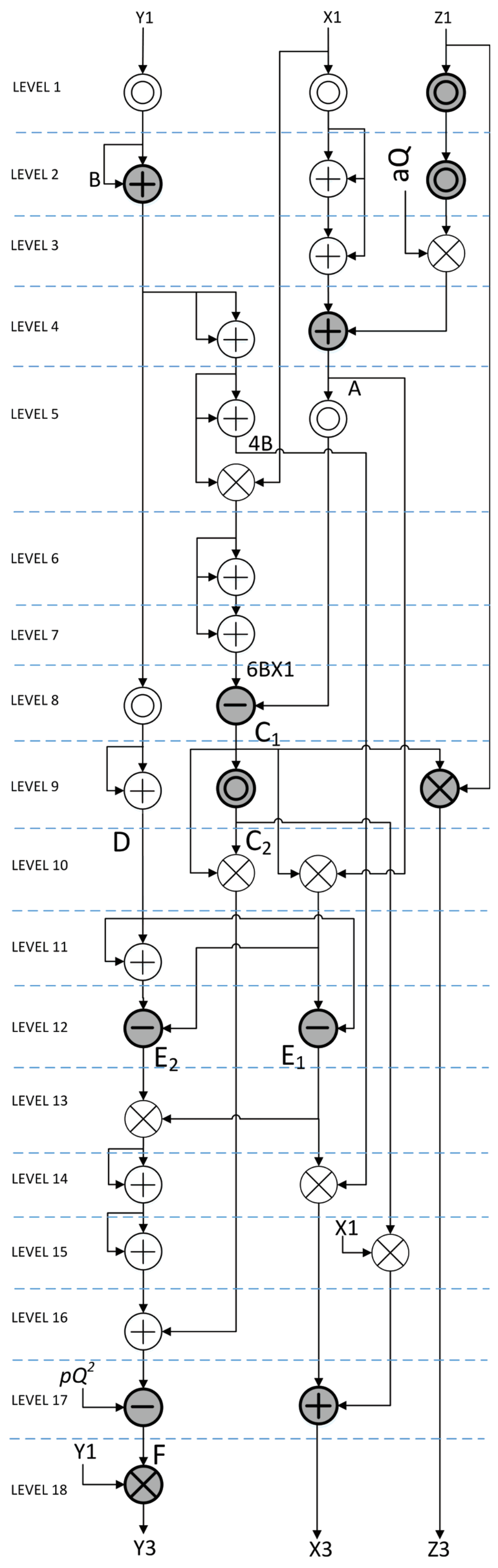


Figure 4.6: RNS point tripling flow diagram on Brainpool256r1

4.3 RNS ECC arithmetic hardware architecture

The general design of RNS ECC arithmetic hardware is illustrated in Figure 4.7. We used two modular reduction units and two modular addition/subtraction units to take advantage of parallel computations and reduce the latency of the system. These primitives are shared between the ECPD and ECPA state machines. The ECPD and ECPA state machines implement and control data flow per clock cycle as shown in Figures 4.1, 4.3, and Figure 4.5 for SECP256K1, ED25519, and Brainpool256r1, respectively. The ECPT state machine is used when implementing the DBC point multiplication algorithm. The ECPT state machine hardware is shared with the ECPD. The data flow diagram of ECPT state machine is shown in Figures 4.2, 4.4, and 4.6 for SECP256K1, ED25519, and Brainpool256r1, respectively.

The Montgomery ladder ECPM in our design performs ECPD and ECPA at the same time. Therefore, the arithmetic primitives cannot be shared between ECPA and ECPD state machines. Figure 4.8 depicts the architecture of the arithmetic hardware to be used in the Montgomery ladder ECPM. Every ECPA and ECPD unit exploits two dedicated modular reductions and modular addition/subtraction hardware units which makes working in parallel possible.

The curves SECP256k1 and ED25519 point arithmetic units are implemented using both SOR_1M and SOR_2M modular reduction architectures. The latency of ECPA, ECPD, and ECPT on the curves SECP256K1 and ED25519 are reported in Table 4.1 for both implementations using SOR_1M and SOR_2M modular reduction units. The Brainpool256r1 field characteristic p is not in the form of pseudo-Mersenne and is not supporting the condition (3.37). Therefore, we used the RNS Montgomery modular reduction to implement point arithmetic units of the curve. The latencies of ECPA, ECPD, and ECPT on the Brainpool256r1 curve are reported separately in Table 4.2. The latencies are reported for implementations on VIRTEX-7 and VIRTEX Ultra Scale+ FPGAs. The clock period used for VIRTEX 7 and VIRTEX UltraScale+ FPGAs is 8.0 ns and 6.0 ns, respectively.

Table 4.1: SECP256K1 and ED25519 point operations latency in nano seconds

Modular reduction architecture		SOM_1M		SOM_2M	
ECC Curve	Device/Operation	VIRTEX-7	VIRTEX US+	VIRTEX-7	VIRTEX US+
SECP256K1	ECPD	1072	737	848	583
	ECPA	2496	1716	1936	1331
	ECPT	1624	1117	1288	886
ED25519	ECPD	1040	715	816	561
	ECPA	1552	1067	1216	836
	ECPT	2016	1386	568	1078

Table 4.2: Brainpool256r1 point operations latency in nano seconds

Modular reduction architecture		RNS Montgomery	
ECC Curve	Device/Operation	VIRTEX-7	VIRTEX US+
Brainpool256r1	ECPD	1280	960
	ECPA	2112	1584
	ECPT	1784	1338

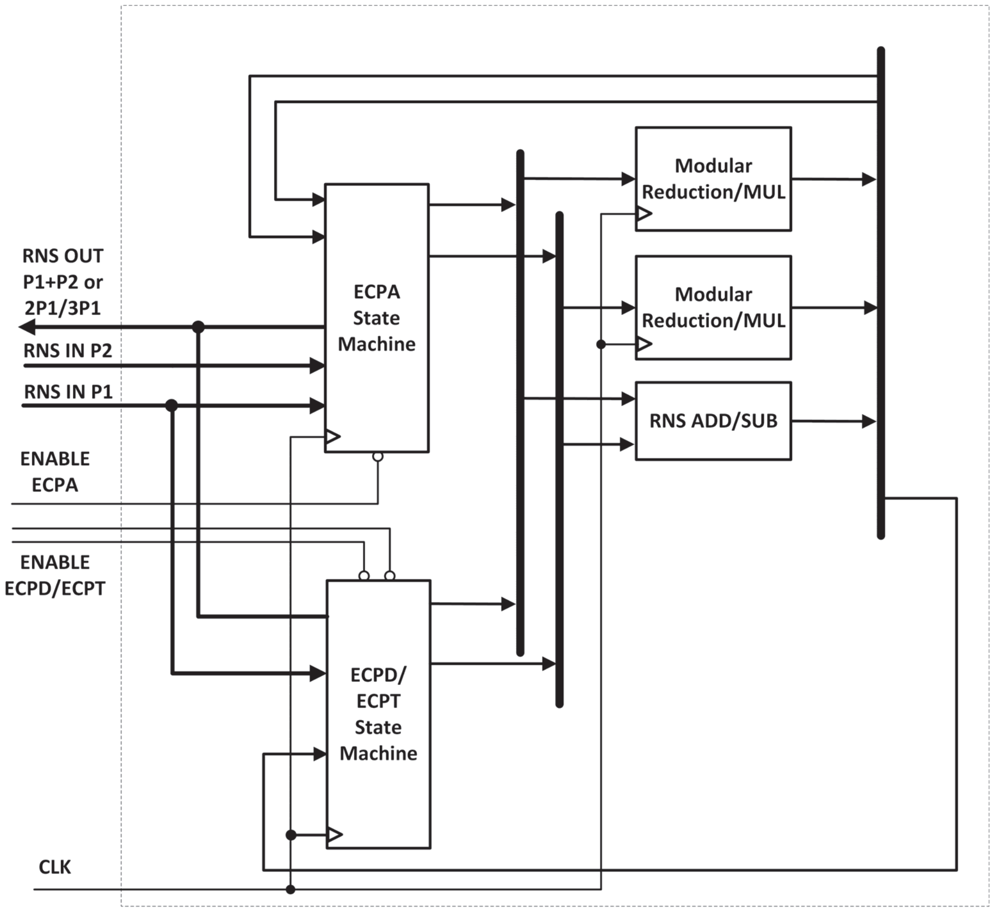


Figure 4.7: General Arithmetic hardware architecture of ECPM

4.4 Point multiplication state machines

The point multiplication state-machine implements the point multiplication algorithms discussed in section 2.2.12. In this research, we implemented Double-and-add, Montgomery ladder, NAF, NAF$_3$, and DBC point multiplication algorithms. We used DBC Tree and DBC L-T algorithms to decode a scalar to its DBC representations. The point multiplication state machine receives a scalar k and a point P on the curve as inputs and then exploits the point arithmetic hardware units to calculate the new point $Q = k \cdot P$. The format of the scalar k depends on the algorithm used for point multiplication. Conversion to the proper format, either NAF, NAF$_3$, DBC, or JSF, is done by software and the result is saved in the FPGA. Figure 4.9 shows the top-level architecture of the point multiplication hardware. In the general design, arithmetic units can be any of the architectures in Figure 4.7 or 4.8 designed for SECP256k1, ED25519, and Brainpool256r1 curves. In the following, we have detailed the state machines architectures used in this research.

4.4.1 Double-and-add scalar multiplication state machine

The Double-and-add algorithm shown in Algorithm 6 is the simplest way to calculate the point multiplication of scalar k. The binary representation of k is $(k_{l-1} \dots k_i \dots k_0)$, where $l = \lceil \log_2 k \rceil = 256$. The hardware implementation of this algorithm is shown in the state machine diagram of Figure 4.10. At state S0, if the RESET signal is not active, the key (scalar k) and

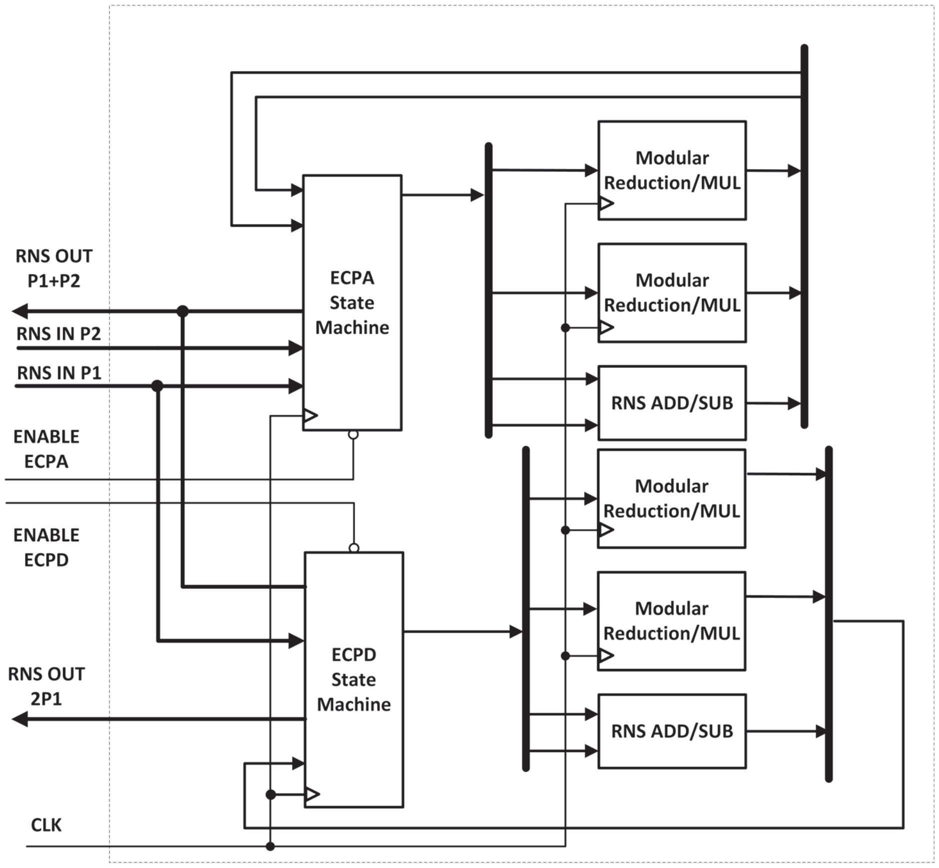


Figure 4.8: Arithmetic unit hardware architecture for Montgomery ladder algorithm

point P are read from the input. the register i is set to value of l. When the 'Ready' signal is zero, it means that the output of the state machine is not valid. The point P is set as one input of the ECPA and is copied to register R. At the State S1, the ECPD input is fed by the register R and the register i is decremented. At the next state, S2, the ECPD starts calculating $2P$ and updates the contents of register R to the new value $2R$. At S3, the ECPD is disabled. Then, the bit k_i is checked. If it is one, a point addition to P must be performed at state S4

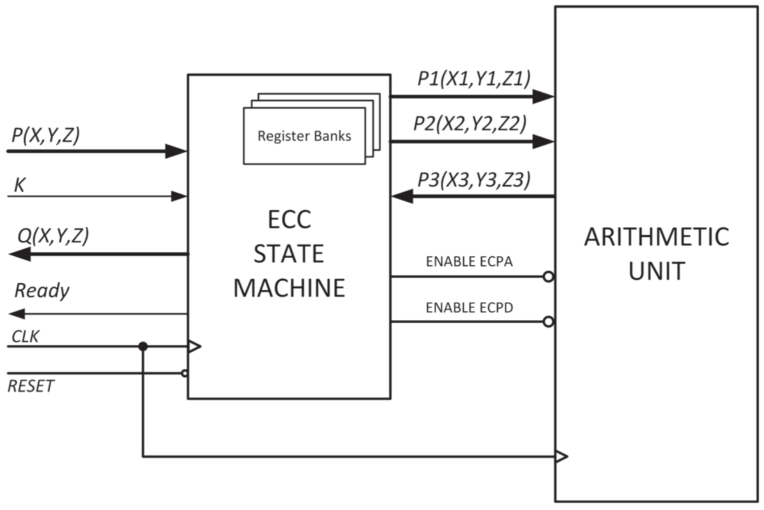


Figure 4.9: RNS ECC Core Hardware implementation

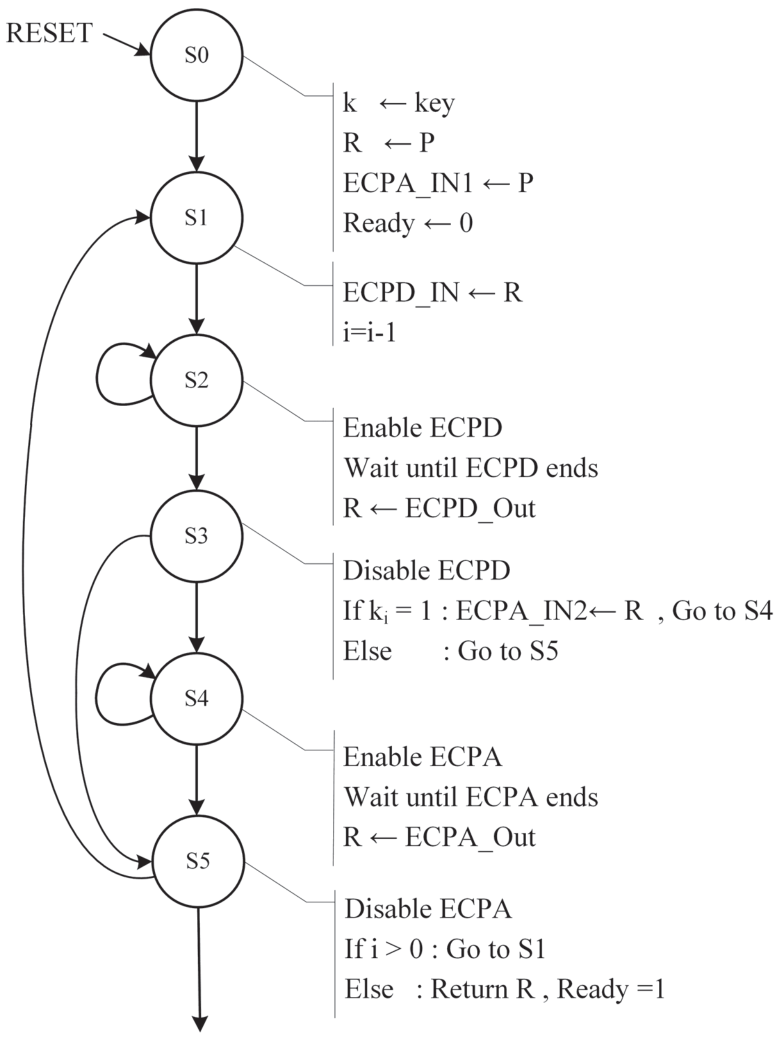


Figure 4.10: Double-and-Add algorithm state machine

and register R is updated to the output of ECPA. Else if the bit K_i is zero, the ECPA step is bypassed to state S5. At state S5, the ECPA is disabled, and if $i > 0$, the state machine goes back to state S1. Otherwise, the point multiplication is completed. The content of register R is set to the output and 'Ready' signal is set to one.

4.4.2 Montgomery ladder scalar multiplication state machine

Montgomery ladder algorithm — shown in Algorithm 7— performs ECPA and ECPD in parallel. The state machine starts state S0 with loading the key to register k, point P to register R_0 and input of ECPD, and setting the 'Ready' signal to zero. At state S1, the ECPD is enabled to calculate $2P$. The value of $2P$ is saved in register R_1. At state S2, the registers R_0 and R_1 fed the inputs of ECPA. The bit k_i is checked. If k_i is one, the content of register R_1 is used as the input of ECPD. Otherwise, R_0 is fed to the input of ECPD. At state S3, both ECPD and ECPA are enabled. At S4, the state machine waits for completion of ECPA. Then, the values of registers R_0 and R_1 are updated at state S5 depending on the value of bit k_i. At S6, The ECPA and ECPD are disabled. If $i > 0$, then register i is decremented, and the state machine rolls over to state S2. Otherwise, the 'Ready' signal will be set to one and register R_0 content is introduced to the output.

4.4.3 NAF method scalar multiplication state machine

In this algorithm, the NAF representation of the scalar k is used with Algorithm 9. Assuming that $\forall i \in [l-1, 0], (k'_{l-1} \dots k'_i \dots k'_0)$, $k'_i \in \{-1, 0, 1\}$. The length of NAF(k) is l. Then each k'_i is mapped to a two-bit binary number as $\{-1 \rightarrow \text{‘}11\text{’}\}$, $\{1 \rightarrow \text{‘}00\text{’}\}$, and $\{1 \rightarrow \text{‘}01\text{’}\}$ to be recognisable by the hardware. The state machine diagram of the NAF method point multiplication is shown in Figure 4.12. The state machine starts at state S0 when the signal RESET is not active. At S0, the decoded NAF of scalar k and point P are read from the input and saved in resisters k and R, respectively. The register i is set to $2\times l$ (as each k_i is mapped to two bits) and the ‘Ready’ signal is set to zero. At state S1, the additive inverse of P, that is, $-P$ is calculated and saved in register S. At state S2, the input of ECPD is set by register R and the register i is decremented by 2. At state S3, the register R is updated by the output of ECPD. At state S4, the ECPD is disabled, register R is set to one of the inputs of ECPA. If k_i = ‘01’ is then the ECPA gets P, if k_i = ‘10’ the ECPA gets S as the second input and the state machine goes to state S5. Else if k_i = ‘00’ then there is no need of performing a point addition and he state machine jumps to state S6. At state S6, if $i > 2$ then the state machine rolls over to S2, otherwise the point multiplication is complete. Register R is set to the output and ‘Ready’ signal is set to one.

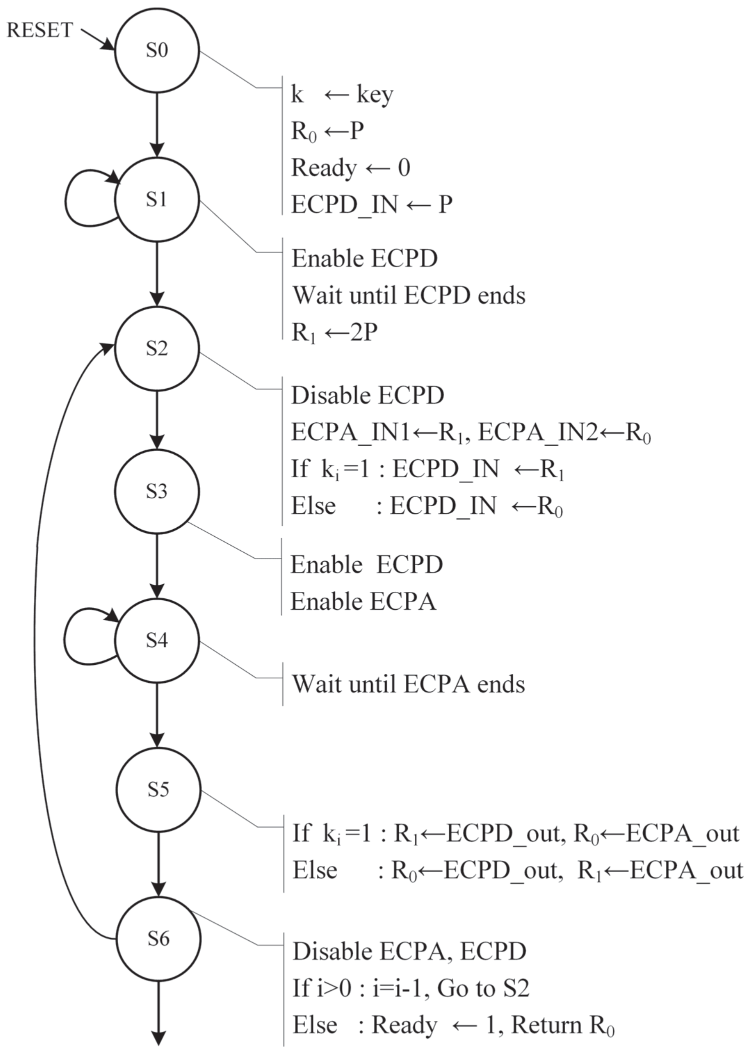


Figure 4.11: Montgomery ladder algorithm state machine

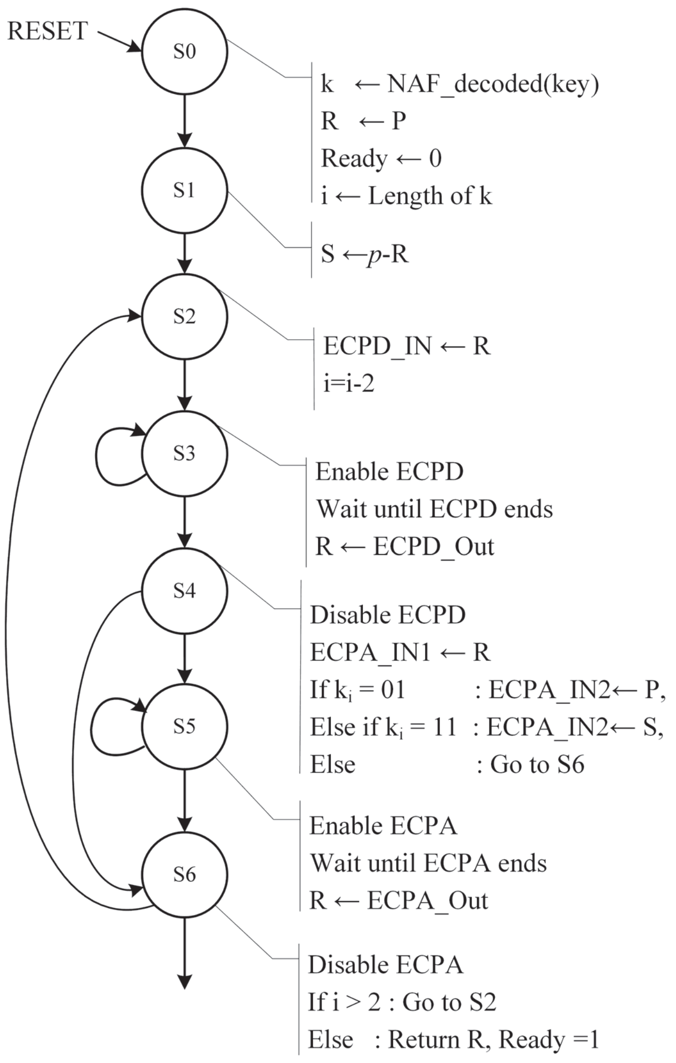


Figure 4.12: NAF algorithm state machine

4.4.4 NAF$_3$ method scalar multiplication state machine

The NAF$_3$ method also utilises Algorithm 9. However, The representation of scalar k in NAF$_3$ is given by $(k'_{l-1} \dots k'_i \dots k'_0)$, $k'_i \in \{-3, -1, 0, 1, 3\} \forall i \in [l-1, 0]$. Where, l is the length of the expansion. Therefore, the points $-P$, $3P$, and $-3P$ must be pre-computed. Each k_i is mapped to a three-bit binary number to be understandable for the hardware. We consider the mapping as $\{-3 \rightarrow \text{‘111’}\}$, $\{-1 \rightarrow \text{‘011’}\}$, $\{0 \rightarrow \text{‘000’}\}$, $\{1 \rightarrow \text{‘001’}\}$, and $\{3 \rightarrow \text{‘101’}\}$. Figure 4.13 shows the NAF$_3$ method point multiplication state machine. At state S0, the NAF$_3$ representation of scalar k and point P are loaded to registers k and $R0$, respectively. The signal ‘Ready’ is set to zero. At states S1 to S5, the points $-P$, $3P$, and $-3P$ are calculated and saved in registers $R1$, $R2$, and $R3$, respectively. At state S5, the register R is loaded by $R0$ or $R3$, depending on the k_{l-1} value. At state S6, the register R is fed to the input of ECPD. At state S7, register R is updated by the output of ECPD. At state S8, register R is fed to one of the inputs of ECPA. The value of k_i determines which one of $R0$ to $R3$ must be used as the other input of ECPA. If k_i = ‘000’ then there is no need of point addition and the state machine skips to state S10. At state S9, ECPA is enabled. The register R is updated by the output of ECPA. At State S10, if $i > 3$, then, register i is decremented by three and the state machine rolls over to state S6. Otherwise, the output is set by the value of register R and signal ‘Ready’ is set to one.

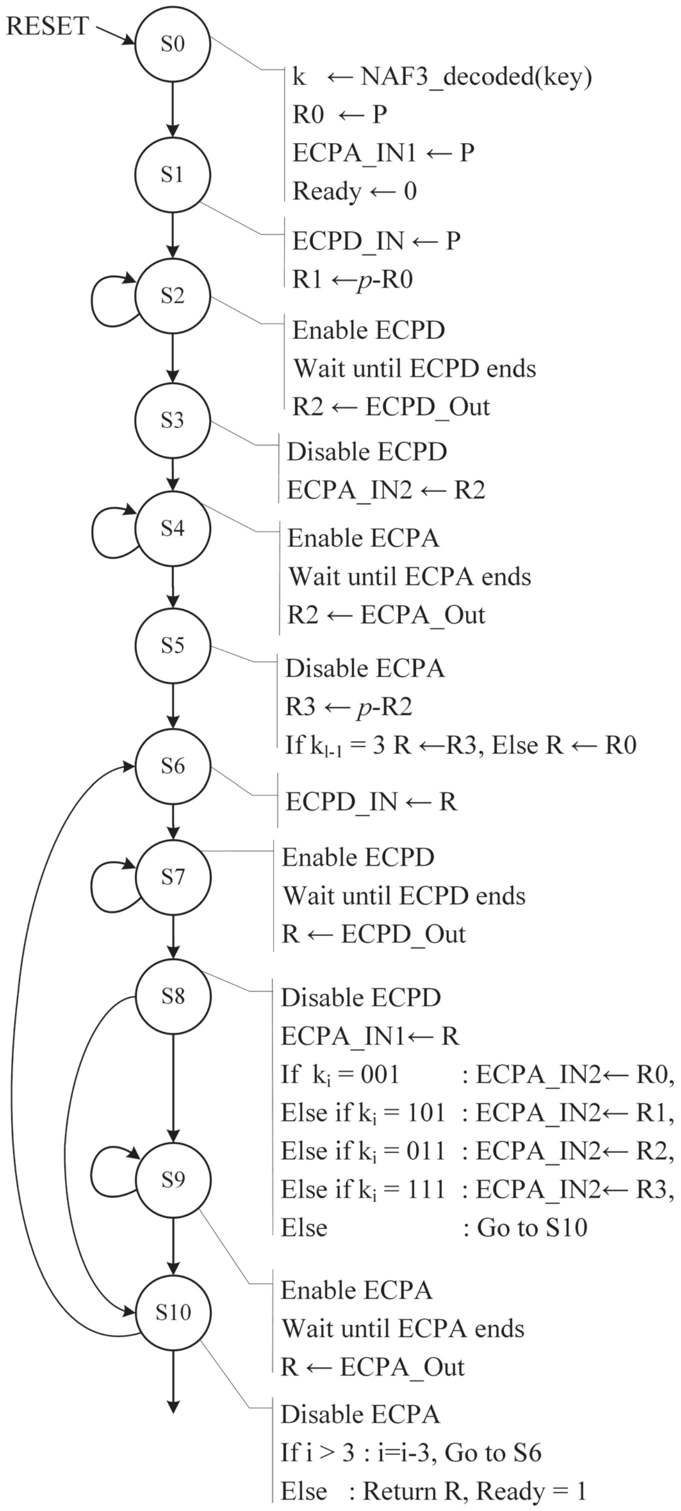


Figure 4.13: NAF_3 algorithm state machine

4.4.5 DBC scalar multiplication state machine

The DBC scalar multiplication exploits the DBC representation of the scalar k. The DBC representation consists of six distinct operations: double, double and add, double and subtract, triple, triple and add, and triple and subtract which are mapped to 3-bit binary numbers ‘011’, ‘101’, ‘111’, ‘010’, ‘100’, ‘110’, respectively. The DBC representation of k is obtained from either the Tree or L-T algorithms discussed in section 6. Figure 4.14 shows the DBC point multiplication state machine diagram. The state machine starts at state S0, after an exit from

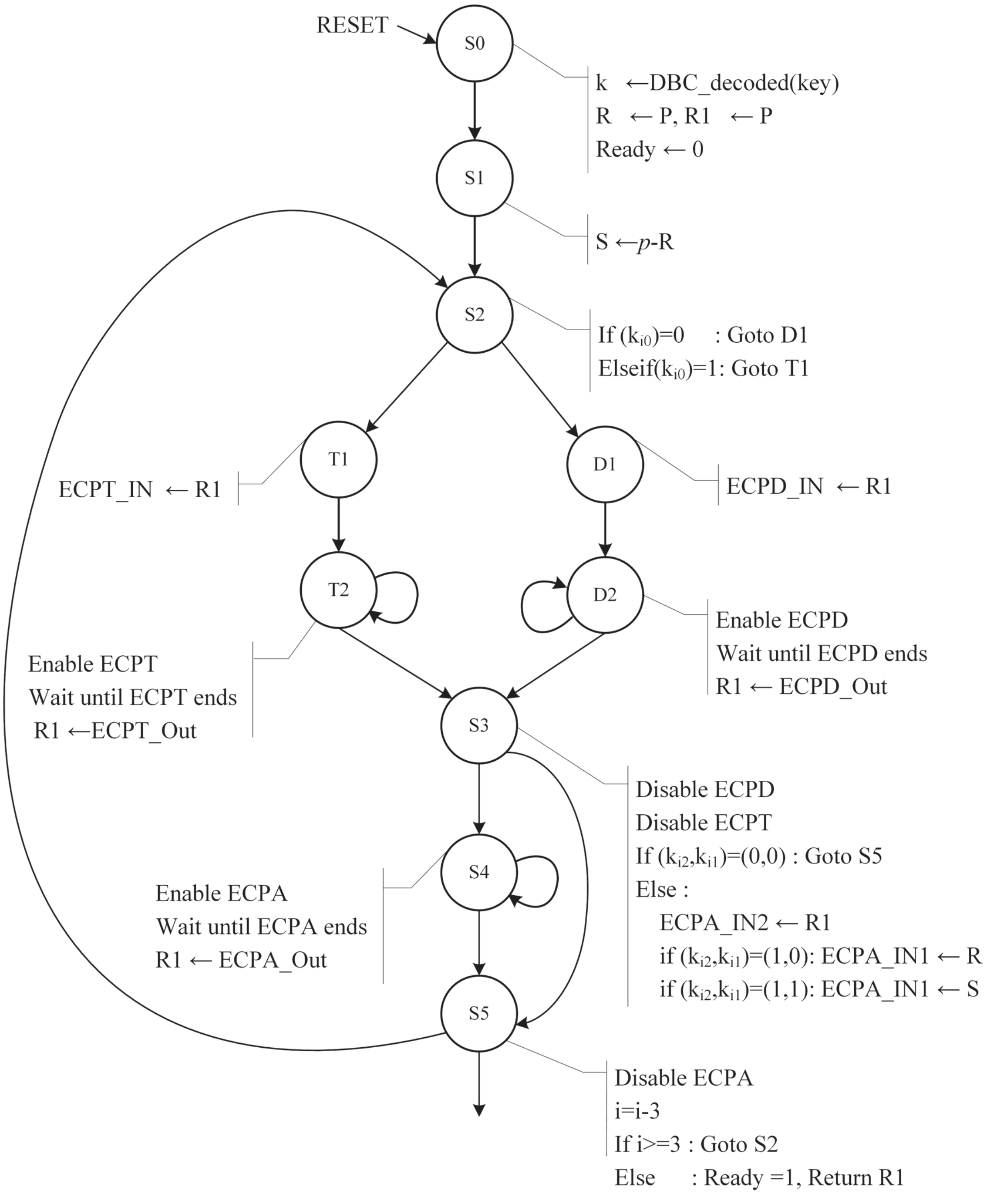


Figure 4.14: DBC algorithm state machine

the RESET state. The mapped DBC value of the scalar is loaded to register k, point P is saved to registers R and $R1$, register i is updated with the bit-length of k, and signal ‘Ready’ is set to zero. At state S1, the additive inverse of P is calculated and saved in register S. At state S2, if bit zero of k_i (denoted by k_{i0}) is zero, then the next state will be D1 where register $R1$ is fed to the input of ECPD to do a doubling at state D2. Otherwise, if the bit K_{i0} is one, then the next state is T1, where the register $R1$ is fed to the input of ECPT to do a point tripling at state T2. At the end of D2 or T2 state, register $R1$ is refreshed by either ECPD or ECPT, respectively. At state S3, ECPD and ECPT are disabled. If the two most significant bits of k_i denoted by (k_{i2}, k_{i1}) are $(1, 0)$, then a point addition to P is done at the next state S4. If these bits are $(1, 1)$ then, a point addition to $-P$ is done at state S4. Otherwise, the state machine skips performing a point addition at S4 and enters state S5. At state S5, register i is decremented by three. If the value of register $i > 3$ then the state machine rolls over to state S2. Else, it outputs the value of register R and sets the signal ‘Ready’.

4.4.6 GLV method scalar multiplication state machine

The GLV method point multiplication is implemented only for SECP256k1 curve which has an efficient endomorphism property. Algorithm 11 shows GLV point multiplication using JSF representation of the scalars k_1 and k_2. If P is a point on SECP256k1 curve, then the point $Q = \lambda \cdot P$ on the curve can be easily calculated using the endomorphism map $\lambda \cdot P(x, y) = (\beta x, y)$. Assuming that $R = k_1 \cdot P + k_2 \cdot Q$, using Shamir’s trick, and JSF representation of k_1 and k_2 we can calculate point R, i.e. $R = \begin{bmatrix} P & Q \end{bmatrix} \cdot \begin{bmatrix} {k_1}^{l-1} & \cdots & {k_1}^{0} \\ {k_2}^{l-1} & \cdots & {k_2}^{0} \end{bmatrix}$. Considering that ${k_1}^i, {k_2}^i \in \{-1, 0, 1\} \forall i \in [l-1, 0]$, we will have nine distinct operations that are a point doubling only, and a point doubling then addition of any of the points P, Q, $-P$, $-Q$, $P + Q$, $P - Q$, $-P + Q$, and $-P - Q$. We pre-compute these eight points and save them in a ROM. The state machine then performs the Shamir’s trick by first a point doubling and then a point addition to one of the pre-computed points read from the ROM. The columns in JSF representation of the scalar k_1 and k_2 are decoded to a 4-bit binary number as follows:

$k_1 \rightarrow$	0	1	1	1	-1	-1	0	-1	0
$k_2 \rightarrow$	1	0	1	-1	-1	1	-1	0	0
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
	’0000’	‘0001’	‘0010’	‘0011’	‘0100’	‘0101’	‘0110’	‘0111’	‘1000’

These 4-bits correspond to the memory locations where the pre-computed points are saved. For example, Q is saved at memory address ‘0000’, $P + Q$ at memory address ‘0010’, and so on. The GLV point-multiplication state machine diagram is shown in Figure 4.15. At state S0, the decoded JSF representation of scalars k_1 and k_2 is read from the input and saved in register k. The ‘Ready’ signal is set to zero. At state S1, the leftmost 4-bits of the register k are used as the ROM address to fetch the corresponding point and load to register R. At state S2, the register R is fed to the input of ECPD. The pointer i is decremented by 4. It is now pointing to the next 4 bits in register k. At state S3, ECPD is enabled. Register R is updated with the output of ECPD. At State S4, the next four bits of k are read, If it is equal to ‘1000’ which corresponds to ${k_1}^i$ = ‘0’ , ${k_2}^i$ = ‘0’, then the state machine rolls over to state S2. Otherwise, it is used as the address of the ROM. The fetched point from the ROM is fed to one of the inputs of ECPA. The second input of ECPA is taken from the register R. At state S5, ECPA is enabled. The state machine stays in this state until ECPA output is ready. The register R is updated by the ECPA output at the last clock cycle of state S5. At state S6, if all bits of register k are not read; the state machine rolls over to S2. Otherwise, the state machine sets the output by the contents of register R and the ‘Ready’ signal to one.

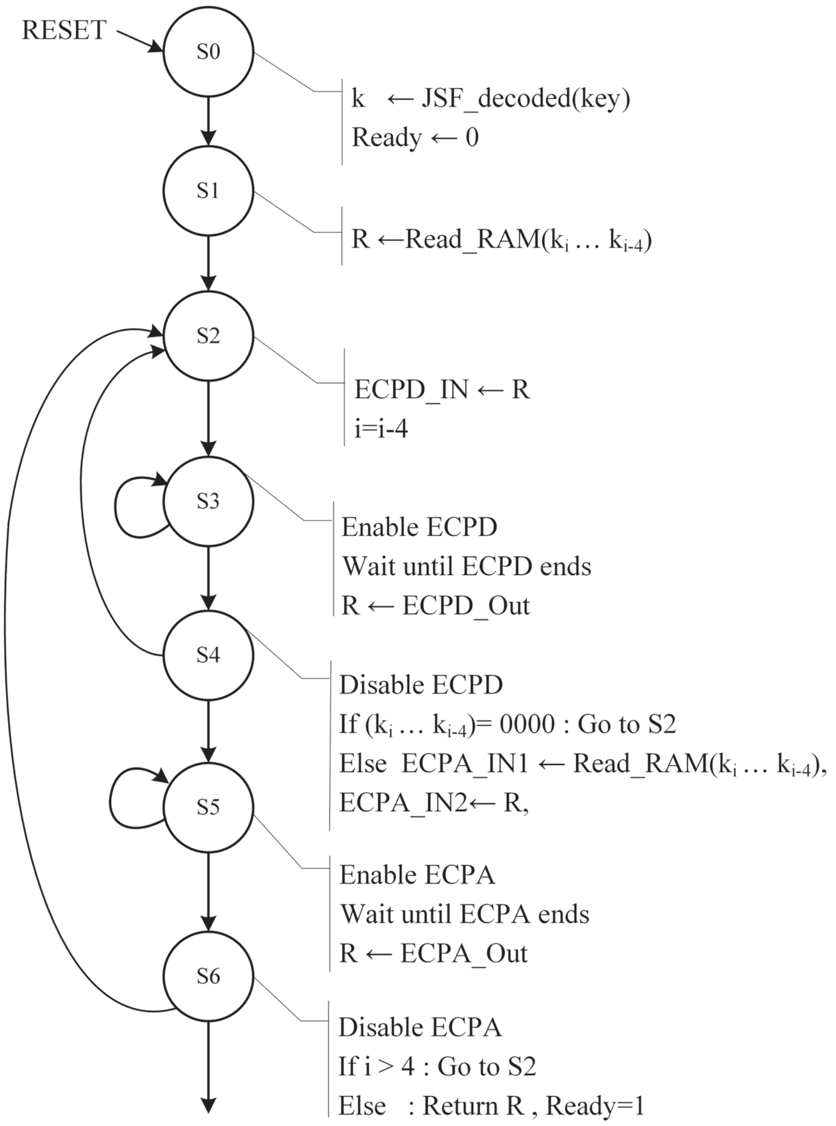


Figure 4.15: Scalar multiplication state machine using GLV algorithm

4.5 Implementation results

We designed point multiplication hardware cores for curves SECP256k1, Brainpool256r1, and ED25519. The general architecture shown in Figure 4.9 was implemented for each curve on Xilinx VIRTEX-7 and VIRTEX UltraScale+™ FPGA platforms that are manufactured with 28 nm and 16 nm process technologies, respectively. The arithmetic hardware for curves SECP256r1 and ED25519 is designed utilising the sum of residues modular reduction unit. Both SOM_1M and SOM_2M architectures are applied in our design to study the performance of the hardware. The Brainpool256r1 point multiplication hardware exploits our proposed RNS Montgomery reduction unit. The point multiplication state machines for SECP256K1, Brainpool256r1, and ED25519 curves are implemented using the double-and-add, Montgomery ladder, NAF, NAF$_3$, and both DBC Tree and DBC L-T algorithms. The GLV algorithm state machine was implemented for SECP256k1 curve only. The top-level architecture depicted in Figure 4.9 utilises the general arithmetic unit configuration shown in Figure 4.7. Using a MAPLE program, we generated 1000 random 255 to 256-bit integers and their representations in NAF, NAF$_3$, DBC, and JSF of k_1 and k_2 to be readable by the point multiplication hardware. The performance of each point multiplication algorithm is measured by averaging the latency. The NAF$_3$ state machine needs to pre-compute and store points. However, it can effectively reduce the overall latency of point multiplications. ECPT has to be implemented to use DBC scalar multiplication algorithms. ECPT adds a large logic area to the system. Nevertheless, DBC scalar multiplication algorithms can effectively reduce the latency. The performance of NAF$_3$ algorithm is slightly better than DBC algorithms.

We also used the GLV algorithm to implement SECP256K1 RNS ECC core. The GLV point multiplication requires pre-computations and storage of eight points in the memory. This algorithm offers the fastest and the highest performance compared to all other methods. However, it is only available for the curves with an efficient endomorphism.

Tables 4.3 and 4.4 compare the average scalar multiplication latency of SECP256k1 and ED25519 curves designed with SOR_1M and SOR_2M modular reduction units. The hardware was implemented on VIRTEX 7 and VIRTEX UltraScale+ FPGAs. For ease of comparison, Figures 4.16 and 4.17 visualise the latency of each point multiplication hardware. The main clock frequency used in this experiment was 125 MHz and 166.7 MHz on VIRTEX-7 and VIRTEX UltraScale+, respectively.

Tables 4.5 and 4.6 report the logic area of RNS point multiplication core implemented using SOR_1M and SOR_2M modular reduction units for curves SECP256k1 and ED25519, respectively. Figures. 4.18 and 4.19 compare the Area$\times$Time parameter as the performance indicator for the implemented designs on SECP256k1 and ED25519 point multiplication hardware.

Our results showed that GLV method improves the performance and speed of SECP256K1 point multiplication by 40% and 47%, respectively compared to Double-and-Add algorithm. The performance of NAF$_3$ is better than DBC however, DBC can improve the latency by 22% and 35% compared to NAF$_3$ and Double-and-Add algorithms, respectively.

The performance of Double-and-Add in ED25519 point multiplication, is close to NAF and NAF$_3$ algorithms. Using DBC method for ED25519 is not promising as its performance and speed is not improved compare to Double-and-Add and NAF.

The latency and area of Brainpool256r1 point multiplication hardware— which is designed with the RNS Montgomery reduction unit— is reported in Table 4.8 on both VIRTEX and VIRTEX UltraScale+ FPGAs. Figures 4.20 and 4.21 show the latency and Area$\times$Timing performance index of Brainpool256r1 point multiplication hardware, respectively on VIRTEX and VIRTEX UltraScale+ FPGAs.

The results showed that the DBC L-T algorithm improves the latency of Brainpool256r1 point multiplication by more than 30% compared to Double-and-Add algorithm. Interestingly, the performance of Double-and-Add is very close to NAF, NAF$_3$, and DBC methods.

The point multiplication hardware which implements the L-T DBC algorithm showed 4.7%, 9.6%, and 3.2% improvement in the core average speed compared to the DBC Tree algorithm implementation on SECP256K1, Brainpool256r1, and ED25519 curves, respectively.

We compared our double-and-add implementation on VIRTEX-7 FPGA series with recent similar works in Table 4.7. Our design notably reduces the timing of a scalar multiplication down to fractions of a millisecond, while the latency of other designs falls within the range of a few milliseconds.

Table 4.3: point multiplication latency on VIRTEX-7

Curve	SECP256K1		ED25519	
Modular reduction	SOR_1M	SOR_2M	SOR_1M	SOR_2M
Scalar Multiplication	latency(ns)	latency(ns)	latency(ns)	latency(ns)
Double-and-add	595397	467547	465576	366193
Montgomery ladder	639600	496576	397288	311720
NAF	495183	388140	400791	314889
NAF$_3$	441191	347087	369661	291124
DBC Tree	409933	323300	379607	297697
DBC L-T	390846	308469	367048	287865
GLV	324025	256069	–	–

Table 4.4: point multiplication latency on VIRTEX UltraScale+

Curve	SECP256K1		ED25519	
Modular reduction	SOR_1M	SOR_2M	SOR_1M	SOR_2M
Scalar Multiplication	latency(ns)	latency(ns)	latency(ns)	latency(ns)
Double-and-add	410076	322714	320337	251733
Montgomery ladder	440365	342036	273263	214435
NAF	340696	270957	275374	216631
NAF$_3$	303694	238870	254816	200227
DBC Tree	282133	222573	261247	204934
DBC L-T	269003	212369	252607	198168
GLV	225052	175855	–	–

4.6 Conclusion

We designed a high-speed RNS ECC point multiplication core for the standard SECP256K1 and ED25519 elliptic curves used in hardware security modules. Using an efficient moduli set, we proposed new architectures for RNS addition and subtraction. Our design relies on two high-speed SOR reduction units shared between ECPA and ECPD (and ECPT when using DBC representation) to allow parallel computing. Two different architectures of SOR modular reductions were used in our RNS ECC design to compare how the performance of modular reduction impacts the overall latency of the system. Different scalar multiplication algorithms

Table 4.5: RNS ECC Point multiplication core area for SECP256K1 on XILINX FPGA

Modular reduction	SOR_1M	SOR_2M
Scalar Multiplication	Slices($\times$ 1000) / KLUT / FF / DSP	Slices($\times$ 1000) / KLUT / FF / DSP
Double-and-add	11.99 / 42.77 / 24782 / 280	12.71 / 45.54 / 25518 / 560
Montgomery ladder	17.90 / 63.98 / 36398 / 560	19.32 / 69.50 / 37134 / 840
NAF	12.31 / 43.83 / 25838 / 280	13.04 / 46.60 / 26574 / 560
NAF_3	12.88 / 45.89 / 26894 / 280	13.36 / 47.65 / 27630 / 560
DBC	14.87 / 52.28 / 34286 / 280	15.65 / 55.24 / 35026 / 560
GLV	13.26 / 46.96 / 29006 / 280	14.01 / 49.78 / 29742 / 560

Table 4.6: RNS ECC Point multiplication core area for ED25519 on XILINX FPGA

Modular reduction	SOR_1M	SOR_2M
Scalar Multiplication	Slices($\times$ 1000) / KLUT / FF / DSP	Slices($\times$ 1000) / KLUT / FF / DSP
Double-and-add	10.37 / 37.18 / 20474 / 280	11.06 / 39.45 / 22760 / 560
Montgomery ladder	15.26 / 54.39 / 31622 / 560	16.48 / 59.11 / 32454 / 840
NAF	11.90 / 42.77 / 23084 / 280	12.77 / 46.10 / 23816 / 560
NAF_3	12.35 / 44.35 / 24140 / 280	13.10 / 47.18 / 24872 / 560
DBC	13.99 / 49.69 / 29948 / 280	14.66 / 52.21 / 30680 / 560

were implemented for the ECC state machine. We showed that combining these algorithms with RNS arithmetic greatly improves the point multiplication latency for both SECP256K1 and ED25519 curves. Our design showed a notable improvement in the speed of elliptic curve point multiplication compared to the most recent similar works at the cost of using more logic resources on the FPGA; however, the performance of our design (Area$\times$Latency) is an outstanding achievement compared to the related works in the literature. We implemented our design on the VIRTEX-7 and VIRTEX Ultra Scale+™ family to show the impact of the layout process technology on the speed.

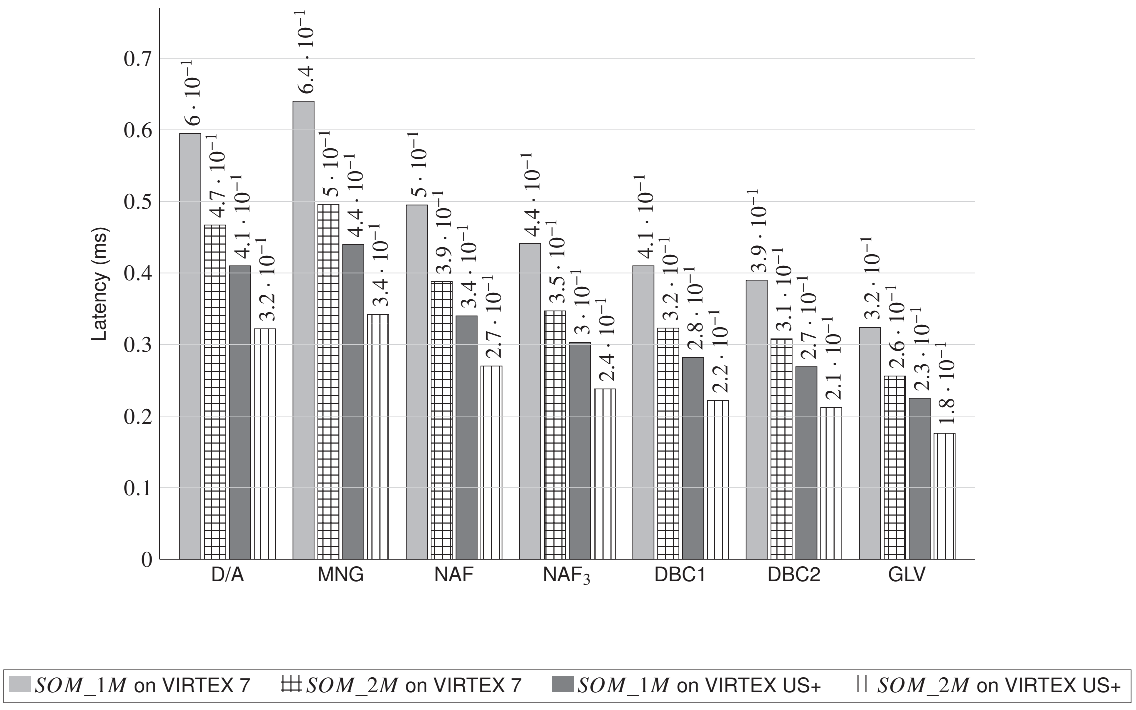


Figure 4.16: SECP25k1 point multiplication latency on Xilinx FPGAs

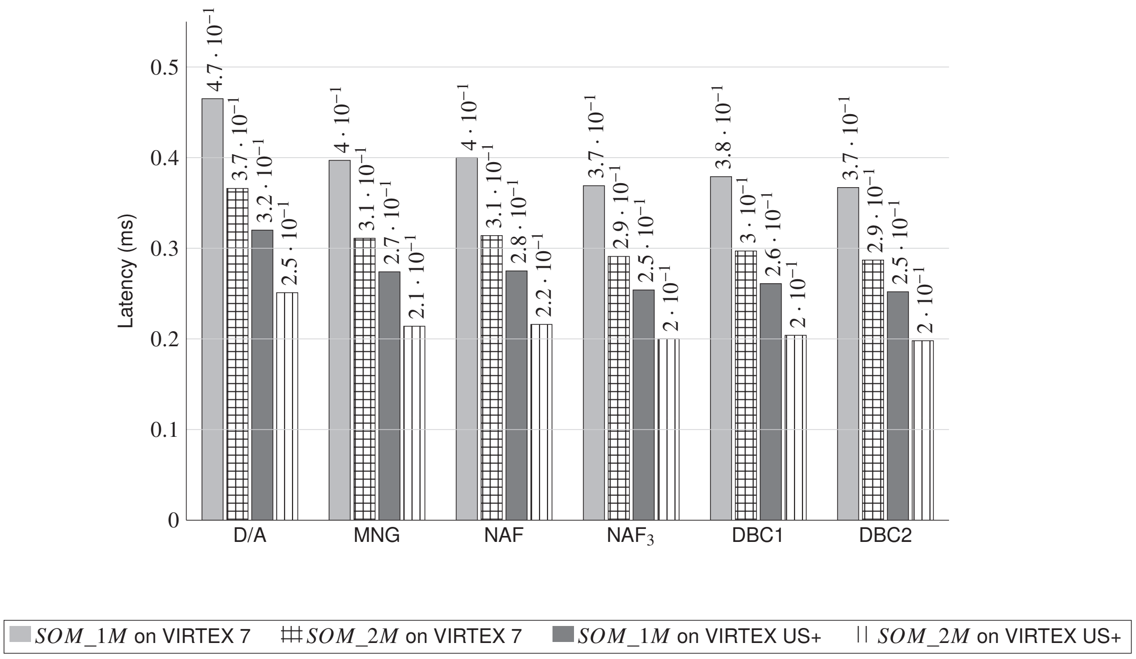


Figure 4.17: ED25519 point multiplication latency on Xilinx FPGAs

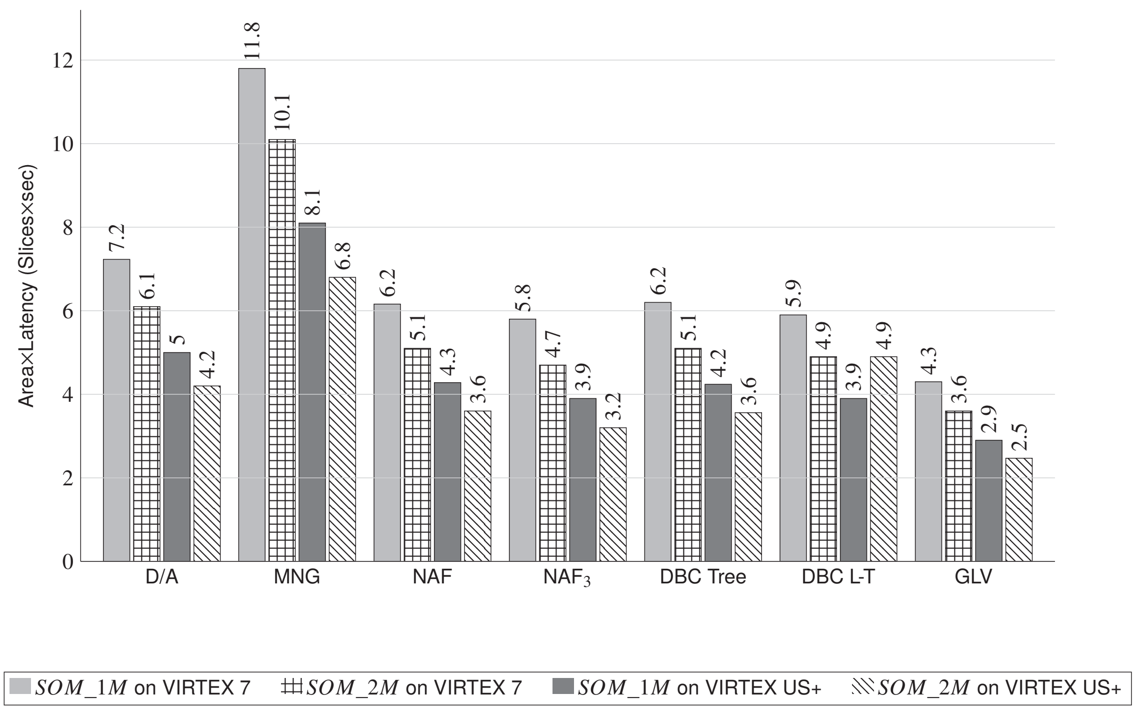


Figure 4.18: SECP25k1 point multiplication performance Area × Timing

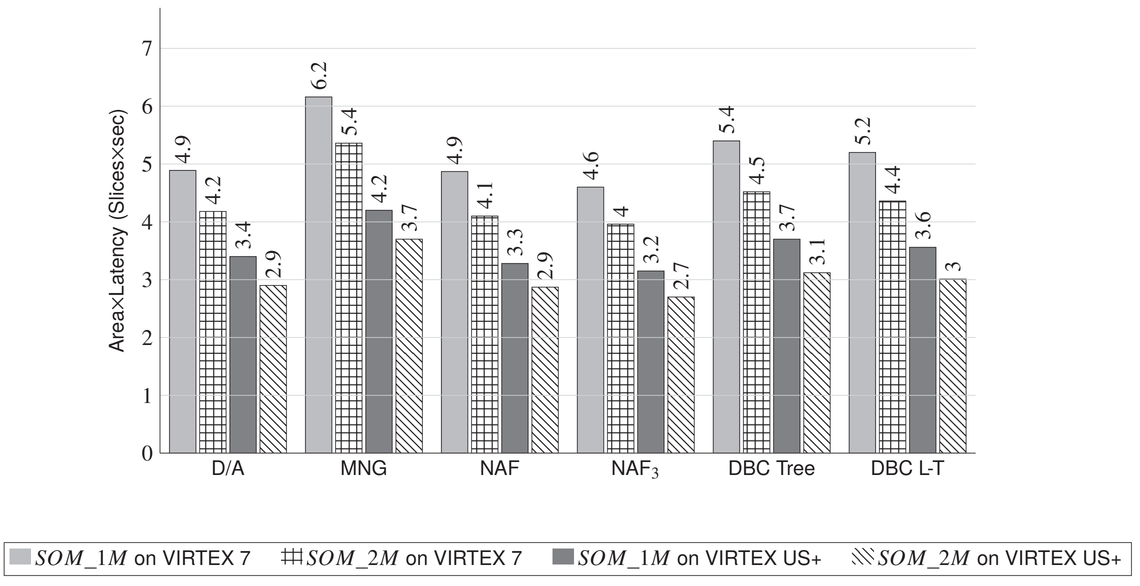


Figure 4.19: ED25519 point multiplication performance Area × Timing

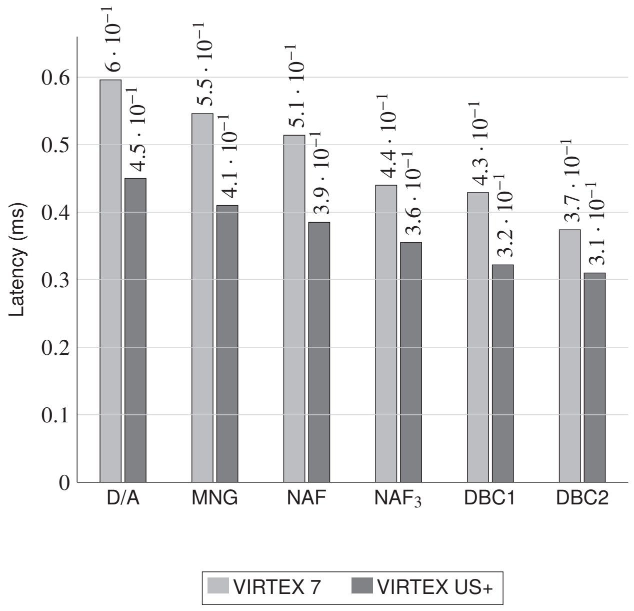


Figure 4.20: Brainpool256r1 point multiplication latency on Xilinx FPGAs

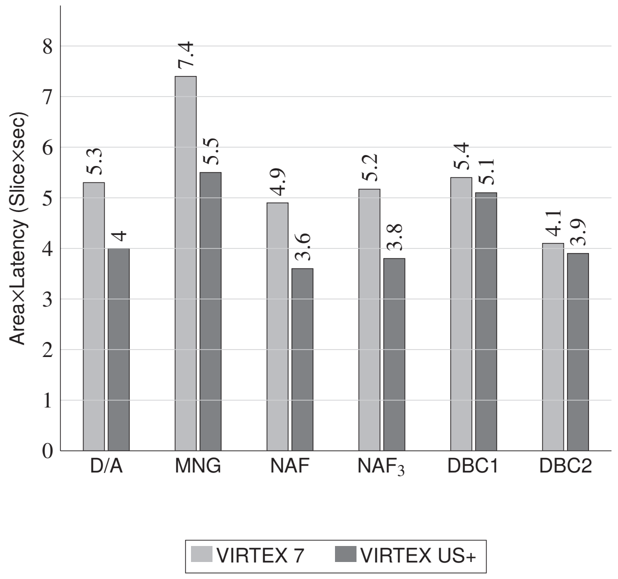


Figure 4.21: Brainpool256r1 point multiplication performance Area $\times$ Timing

Table 4.7: Similar RNS ECC implementations over $\mathbb{F}_p$, p: 256-bit prime

Design	Platform	Area SLICE / KLUT / FF / DSP / BRAM	Latency@Clk Freq ms@MHz	performance Slice×sec
Ours, D/A SECP256K1[109]	VIRTEX-7	12.71K / 45.5 / 25518 / 560 / 0	0.46@125	6.1
Ours, D/A Brianpool256r1	VIRTEX-7	8.8K / 31.2 / 18964 / 28 / 0	0.45@125	5.3
Ours, GLV SECP256K1 [109]	VIRTEX-7	14.01K / 46.9 / 29742 / 560 / 0	0.25@125	3.5
Asif, 2018 [108]	VIRTEX-7	N.R.[1]/ 18.8 /N.R. / N.R. / N.R.	0.73@86.6	14.0 [2]
Asif, 2017 [112]	VIRTEX-7	N.R. / 96.9 /N.R. / N.R. / N.R.	2.96@72.9	71.7[2]
Matutino, 2017 [113]	SPARTAN-6	789 / 1.988 / 786 / 6 / 26	25.4@147	20.1[2]
Alrimeih, 2014 [114]	VIRTEX-6	11.2K / 32.9 /N.R. / 289 / 128	20.5@100	229.6
Esmaeildoust, 2013 [115]	VIRTEX-2	N.R. / 28.7 / N.R. / N.R. /N.R.	12.39@50.2	88.8[2]
Guillermin, 2010 [103]	Stratix II	9.17K ALM /N.R. / N.R. / 96 / N.R.	0.68@157.2	6.2
Schinianakis, 2009 [116]	VIRTEX-E	N.R. / 32.716 / N.R. / N.R. / N.R.	82.9@39.7	677.7[2]
Yuan, 2007 [117]	VIRTEX-2 PRO	41.59K / N.R. / N.R. / N.R. / N.R.	2.66@94.7	110.6

[1]N.R.: Not Reported.

[2]Slices are estimated as KLUT/4.

Table 4.8: RNS ECC point multiplication Latency and Area for Brainpool256r1 on Xilinx FPGA

Platform	VIRTEX 7		VIRTEX UltraScale+	
Scalar Multiplication	Latency ns	Area Slices(×1000) / KLUT / FF/ DSP	Latency ns	Area Slices(×1000) / KLUT / FF/ DSP
Double-and-Add	595306	8.8 / 31.2/ 18964 / 128	450010	8.8 / 31.0 / 18958 / 128
Montgomery ladder	546264	13.6 / 48.4 / 28592 / 256	409470	13.6 / 48.1 / 28344 / 256
NAF	514772	9.5 / 33.8 / 20648 / 128	385655	9.5 / 33.7 / 20542 / 128
NAF$_3$	474103	10.9 / 39.5 / 21176 / 128	355360	10.9 / 39.1 / 21074 / 128
DBC Tree	429783	12.5 / 45.1 / 24294 / 128	322337	12.5 / 44.8 / 24253 / 128
DBC L-T	413891	12.5 / 45.1 / 24294 / 128	310418	12.5/ 44.8 / 24253 / 128

5
Side-channel power analysis of the GLV RNS ECC

5.1 Introduction

In this chapter, we first introduce a new RNS GLV ECC co-processor design in section 5.2, which is not using parallel computation, with an aim to reduce the area of the core logic. This co-processor is then used for side-channel data analysis. Our first motivation to choose RNS GLV ECC co-processor for side-channel analysis is that the RNS has been used as a countermeasure to side-channel attacks in some literature such as [118] and [119]. Next, as illustrated in Figure 4.15, the GLV state machine reads different pre-calculated points from a ROM. A reading from ROM is a low-power operation in comparison to the arithmetic operations performed in an ECC co-processor. It is presumed that its side-channel data become lost in the presence of the white-noise generated by other hardware resources. Consequently, all the read from ROM operations may look very similar and may not be distinguished from their side-channel power spectrum. Therefore, the combination of the RNS and the GLV method is expected to be a powerful countermeasure to side-channel attacks. Section 5.3 describes our experimental setup, the configuration of the RNS GLV co-processor to operate with a set of private keys, and side-channel power data collection. In section 5.4, we present our observations of applying machine-learning and deep-learning techniques on the side-channel power data.

5.2 Design of a low-cost RNS GLV ECC

The low-cost design of the RNS GLV point multiplication core consists of an arithmetic unit with an ECPD and an ECPA state machine that share one SOR_2M modular reduction unit, one Modular multiplication unit, one addition, and one subtraction unit. The hardware architecture is illustrated in Figure 5.1. The data flow in ECPD and ECPA state machines is shown in Figure 5.2, which implements curve SECP256K1 group operations. However, unlike the architecture in Figure 4.1, no parallel computation is performed in this design. The ECPD and ECPA state

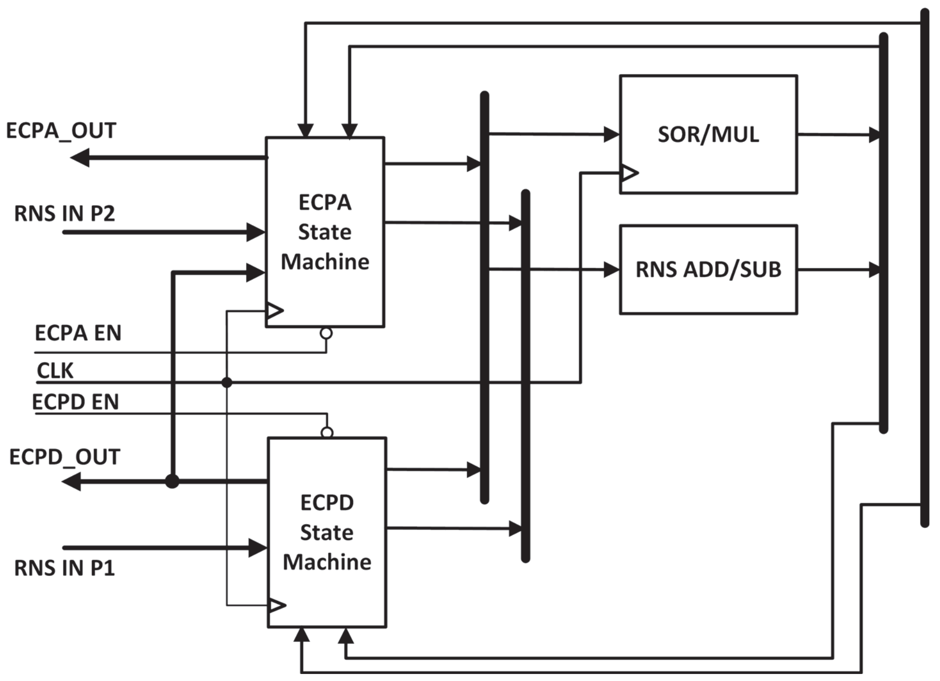


Figure 5.1: RNS GLV core arithmetic unit architecture.

machines are done in eight and sixteen logic levels, respectively using one modular reduction unit. The block diagram of the low-cost RNS GLV ECC point multiplication core is shown in Figure5.3. A set of 16 scalars (private key pairs k_1 and k_2) in JSF format are pre-computed and saved in a separate distributed ROM. The same method described in section 4.4.6 is used to map JSF presentation of the key pairs to a code understandable for hardware. Each key pair is then saved as a 512-bit binary digit in the ROM. These 16 key pairs are chosen such that when the RNS GLV core runs the point multiplication algorithm, our desired training and test side channel power data is generated. More details are presented in Section 5.3.2. The point P, — the generator of the curve SECP256k1 — the pre-computed points $Q = \lambda \cdot P$, $-P$, $-Q$, $(P+Q)$, $(P-Q)$, $(-P+Q)$, and $(-P-Q)$ in their RNS presentation are saved in a ROM that is addressed by the ECC state machine. Recalling that the RNS is a non-positional numbering system, statistically, the hamming weight of the pre-computed points is approximately equal to $\frac{M}{2}$. Where M is the dynamic range of the RNS used in this research; that is $M = 528$. The value of Z_1 is set to $2^{33} - 1$ for all the eight RNS channels to preserve the hamming weight equal to $\frac{M}{2}$ for the Z coordinate. The red dots on Figure 5.4 show the locations where the distributed ROM cells —containing the pre-calculated points — are implemented in the FPGA. It can be observed that the ROM cells are aligned at the centre of FPGA. Therefore, we anticipate having a homogeneous power consumption when readings different points from the ROM. The ECPD and ECPA units use shared arithmetic hardware resources, as illustrated in Figure 5.1. The SOR unit utilises SOR_2M architecture detailed in section 3.7. The flow diagram of the ECPD and ECPA is shown in Figure 5.2. The logic levels present operations that are performed during a modular reduction. The latency of ECPD and ECPA operations are 155 and 370 clock cycles, respectively. After exit from a reset state, the state machine reads one of the sixteen 512-bit keys from the ROM1. (Recalling from 4.4.6, every key-pair bit $({k_1}^i, {k_2}^i)$ is decoded to a 4-bit binary. Therefore the 128-bit keys k_1, and k_2 are decoded to 512-bit binary). At the next state, S1, the state machine sets the address of ROM2 from the leftmost four bits of the key and selects SELD = '0'. At state S2, ECPD is enabled and performs a point doubling. At state S3, based on the value of next four bits of the key, the

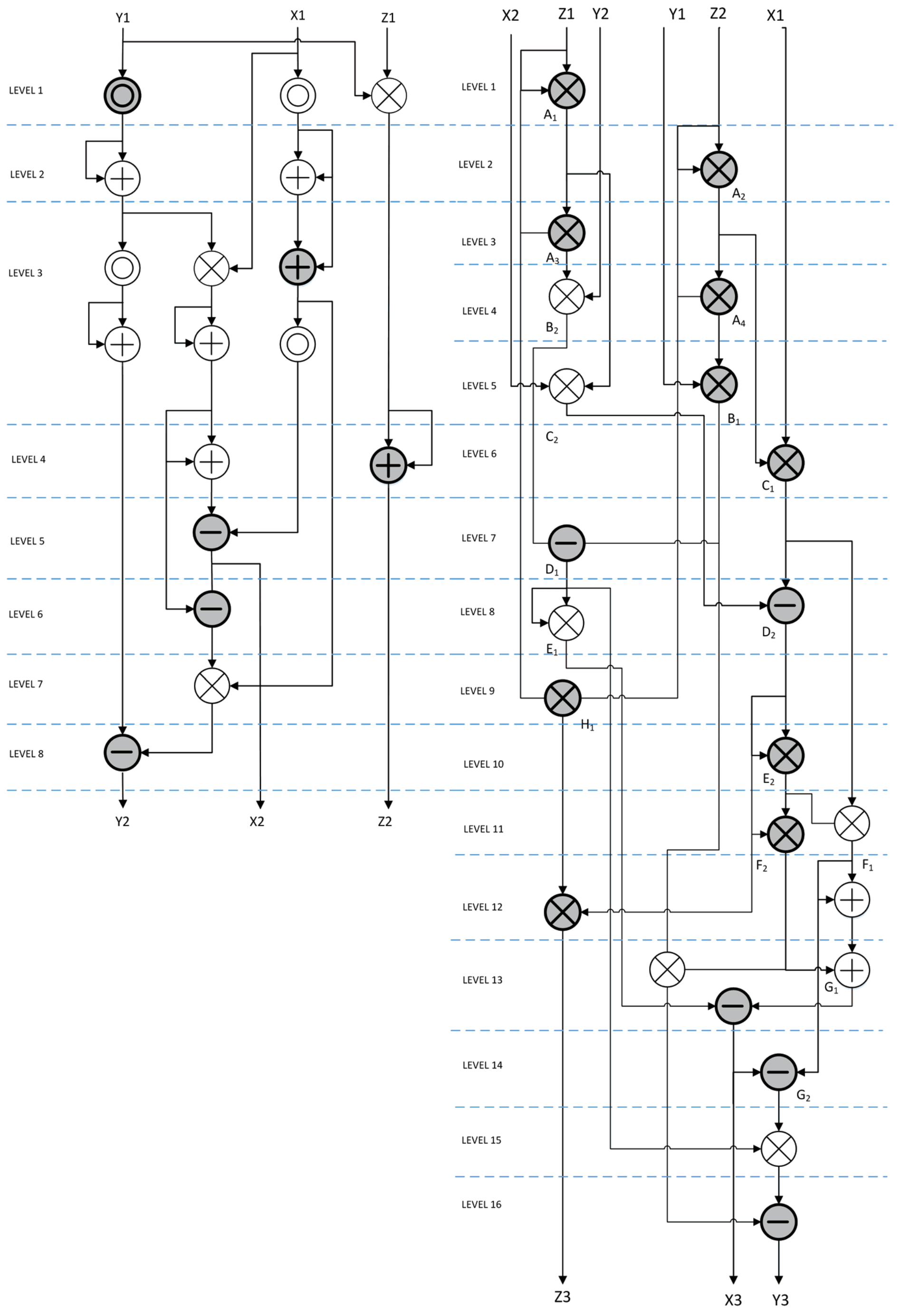


Figure 5.2: ECPD (left) and ECPA (right) flow diagram in the RNS GLV core.

state machine decides either to read a new point from ROM2 and perform a point addition or to perform another point doubling by setting SELA =‘0’ and SELD=‘1’. The state machine then continues this loop until all bits of the key are read. At the final cycle, the result is set to output and the ‘Ready’ signal is set to one.

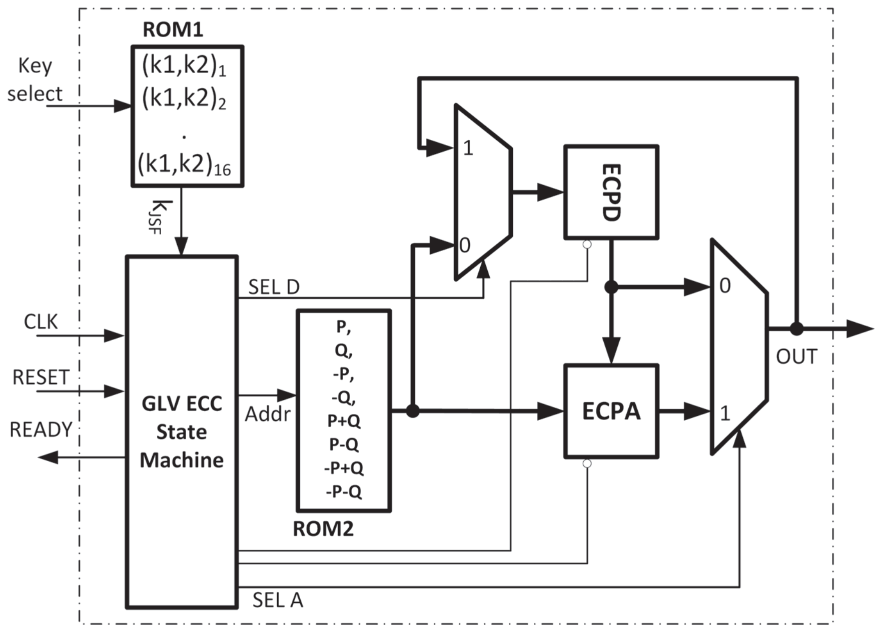


Figure 5.3: RNS GLV ECC point multiplication design.

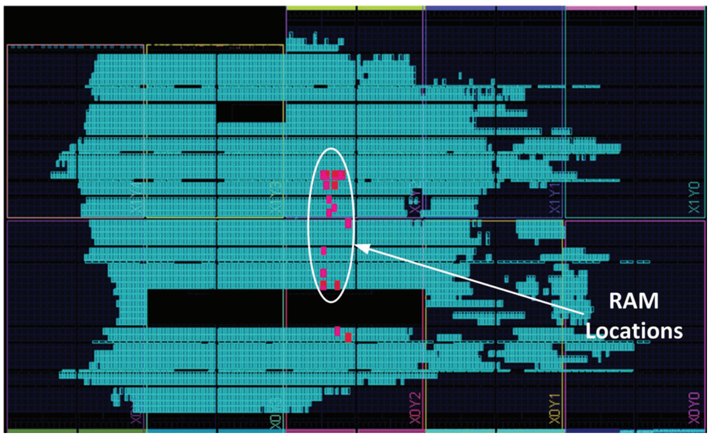


Figure 5.4: Distributed ROM locations on KINTEX-7 FPGA.

5.3 Side-channel data analysis of the GLV ECC core

As discussed in Chapter 2, side-channel attacks are closely related to the existence of physically observable phenomena caused by the execution of computing tasks in the hardware. This type of attack is passive and impalpable. In this research, we focused on power consumption data leakage that is considered side-channel power data in practical attacks. Using machine-learning and deep-learning methods, we studied the side-channel power data acquired by monitoring the current sunk by the FPGA when performing point multiplication operations.

Based on Kerckhoff's principle, it is assumed that the adversary has complete knowledge of the algorithm used for encryption, all related parameters, except the private key, and the required tools to gather side-channel data.

Several sophisticated leakage models are determined by the adversaries to simulate or to improve the attack efficiency. For example, Hamming distance model, studied in our other work in [46] assumes that when a value ’0’ or ’1’ switches into a value ’1’ or ’0’, the actual side-channel leakages are correlated with the Hamming distance of these values. In the following, we introduce our experimental setup and design of our side-channel attack model based on the PoI (Points of Interest) method.

5.3.1 Experimental setup

For the side-channel leakage experiments, we have collected data for the RNS-ECC implementation on KINTEX-7 FPGA, mounted on Sakura-X [120], using Tektronix MDO series oscilloscope with 1GHz bandwidth and 5GS/s sampling frequency. We reduced the FPGA’s clock frequency to 3.0 Mhz, to capture adequate side-channel power signal samples within a clock cycle. The oscilloscope’s sampling rate is set to 25 MS/s such that 8.33 samples can be taken at every clock cycle, which is sufficient to monitor power signal changes in one clock cycle. Sakura-X is a specialised board designed for side-channel leakage data acquisition. It provides a connection to capture the power consumed by the FPGA core through a resistor connected in series. We collected 1600 (100×16) power traces in total. Out of which, 100 traces — each of one million (1M) samples long — are collected for each address value (0-15) of ROM B — that contains the keys k_1 and k_2. Further processing is done to extract the samples of interest from the 1M samples. Traces which relate to the ROM address 0 to 8 are used for training and traces related to address 9 to 15 which are random scalars are used for testing the model accuracy. The data collection process is automated by developing a stand-alone interface application, which requires just initial input parameter settings from the user. The data collection process is classified into the following steps.

- Step 1 - Take input parameters (number of samples, key length) from the user;
- Step 2 - Configure the oscilloscope using MATLAB libraries;
- Step 3 - Send the ROM address of the key from PC to FPGA through the control unit interface;
- Step 4 - Send the trigger signal to start encryption on FPGA;
- Step 5 - Initiate the process of leakage data collection and store in the data file;
- Step 6 - After FPGA completes ECC processing, receive the encryption output and store in a file.

For machine learning analysis, Python along with Keras libraries has been used.

5.3.2 Methodology

As discussed, The GLV ECC core performs an “ECPD only” that corresponds to the scalar JSF pair bits $(0, 0)$, and an ECPD followed by an ECPA in other cases. The point addition uses the output of point-doubling as one input and one of the pre-computed points stored in the ROM2 as the second input. Therefore, the core activity can be categorised into nine distinct operations. That is double, and double then add to any of the points P, Q, $-P$, $-Q$, $P + Q$, $P - Q$, $-P + Q$, or $-P - Q$. The X, Y, and Z coordinates of these points are stored in the form of a 528-bit RNS value. Every 66-bit represents one RNS channel. The hamming weight of all stored values is $\frac{M}{2}$ on average. As a result, the power consumption of a ROM read operation for all the eight saved points are very similar and hard to differentiate. The maximum power is used by the Modular reduction unit. Figure 5.5 shows three randomly chosen power traces related to a modular reduction operation. The peaks are related to RNS multiplication and addition operations. With different inputs, the modular reduction unit power traces have very similar patterns. Figure 5.6 shows power consumption trace of an ECPD followed by ECPD, and ECPD followed by an ECPA adding points P and $P+Q$. An ECPD

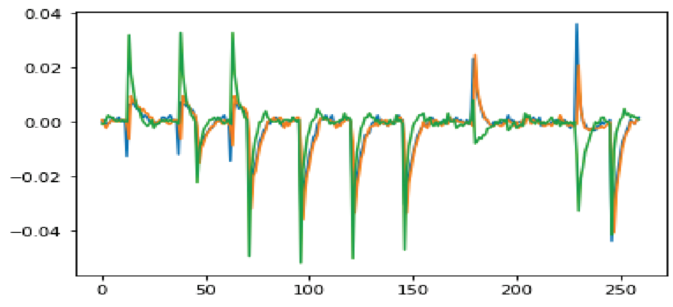


Figure 5.5: Three side-channel power traces of a modular reduction operation on FPGA

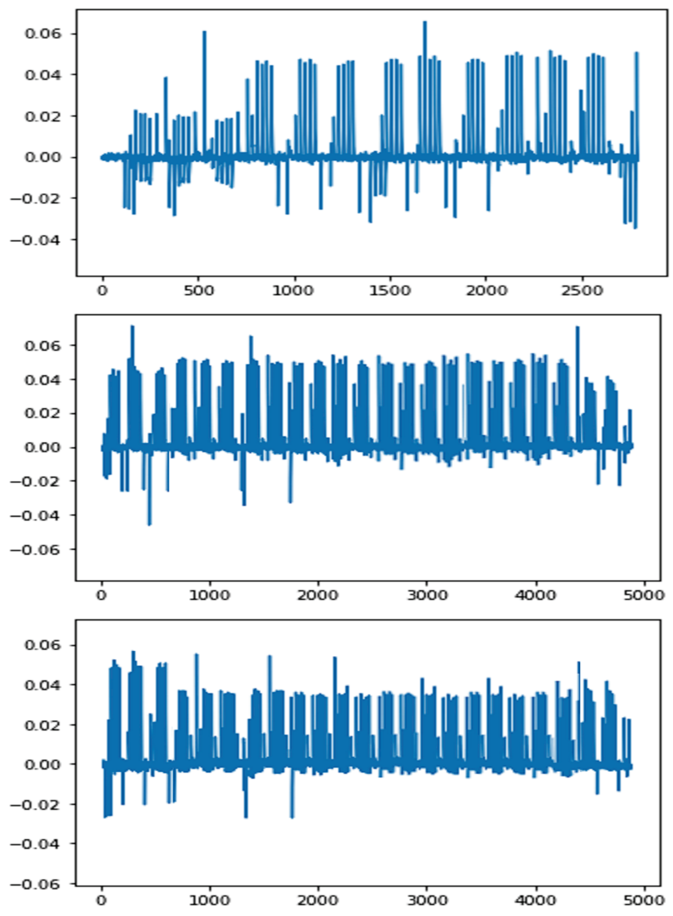


Figure 5.6: side-channel power trace top: two doubling, middle: doubling then addition with P, down: doubling then addition with P+Q

followed by a ROM reading and ECPA operations will take 527 clock cycles. Then, its power trace has 4392 samples in our dataset. The points of interest (PoIs) are samples related to the end of the ECPD, ROM reading and the start of the ECPA. Thus, for the selection of features, we practised two approaches. First, we took a window of 2600 samples that covers two consecutive ECPD operations or one ECPD and a part of ECPA operation. Figure 5.7 shows the power data window used as the machine learning dataset. The rest of the power data is not considered in the dataset preparation since we need to distinguish whether an ECPD operation is followed by an ECPD or an ECPA operation. Moreover, we have to distinguish which point is read from the ROM before performing an ECPA. When an ECPD operation is followed by an ECPD, no data is read from the ROM. In the second approach, we narrowed down the window size and use 192 sample points before a ROM reading 16 sample points of ROM reading and 192 sample points after the ROM reading; in total, the window contains 400 samples. The PoI data is located between ECPD and ECPA power trace. If

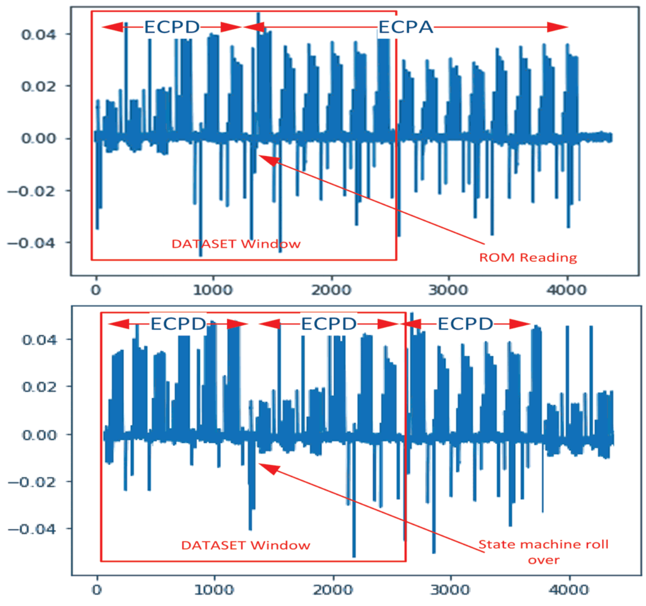


Figure 5.7: Dataset window, top: ECPD-ECPA, down: ECPD-ECPD sequence.

we can distinguish the power patterns of the RAM readings, then it is possible to find the pair of bits of k_1 and k_2. For this reason, we need to train our machine learning algorithms with some distinct key-pairs. We classify nine 128-bit key-pair $L1$ to $L9$ as shown in (5.1) and let the hardware core run by all of these key-pairs. The observed power signals are used to construct our training dataset.

$$
\begin{aligned}
& \qquad\quad \overbrace{}^{128bits} \\
L1 &= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix} \\
L2 &= \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix} \\
L3 &= \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix} \\
L4 &= \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & -1 & -1 & \cdots & -1 \end{bmatrix} \\
L5 &= \begin{bmatrix} 1 & -1 & -1 & \cdots & -1 \\ 1 & -1 & -1 & \cdots & -1 \end{bmatrix} \\
L6 &= \begin{bmatrix} 1 & -1 & -1 & \cdots & -1 \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix} \\
L7 &= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & -1 & -1 & \cdots & -1 \end{bmatrix} \\
L8 &= \begin{bmatrix} 1 & -1 & -1 & \cdots & -1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix} \\
L9 &= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}
\end{aligned}
\tag{5.1}
$$

The side-channel power consumption data of each training key ($L1 - L9$) is sampled 100 times. As a result, by dropping out the first operation for each label (training key-pair $L1 - L9$), we have 12700 sample frames. Each frame includes 2600 features, as depicted in Figure 5.7. We build our dataset by randomly choosing five hundred frames from each class of data ($L1 - L9$). Consequently, we have 4500 data frames that are classified into nine classes. The raw samples are scaled to the range $[-1, 1]$. The absolute of scaled power trace ranged in $[0, 1]$ is used as the input of machine learning algorithms. Figure 5.8 illustrates changes in a sample power signal.

5.4 Applying Machine learning algorithms on side- channel power data

5.4.1 Simple Machine-learning algorithms

In this experiment, we applied SVM, kNN, and Radom Forest algorithms for classification of the power signals. Neither of the algorithms were able to classify the features accurately. The maximum accuracy obtained from the SVM is limited to 22%. Figure 5.9 shows changes in accuracy over changing the SVM algorithm hyper-parameters γ and C. As depicted in Fig 5.10, The application of Random Forest algorithm on our dataset did not provide a reasonable accuracy over a range of hyper-parameters. Our experiment with kNN algorithm resulted in an accuracy of less than 22%. It means that the power signal dataset cannot be classified with these algorithms properly.

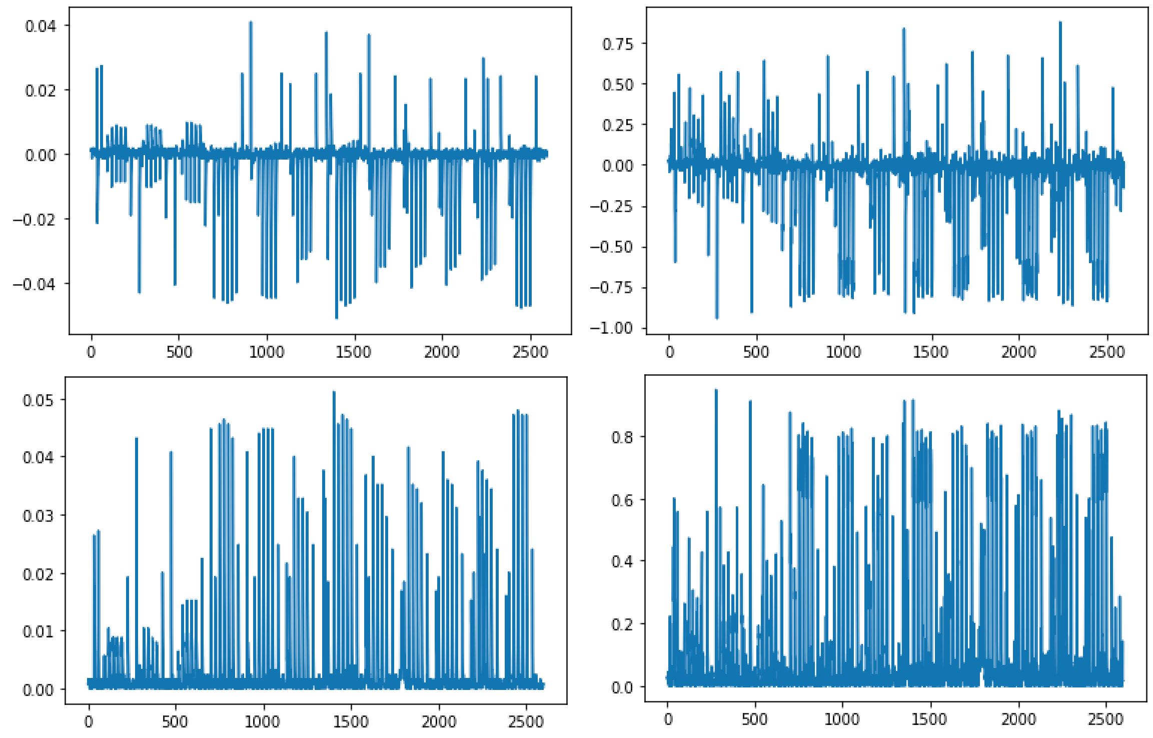


Figure 5.8: Raw power signal (top left), Scaled power signal (top right), absolute power signal (bottom left), and absolute scaled signal (bottom right).

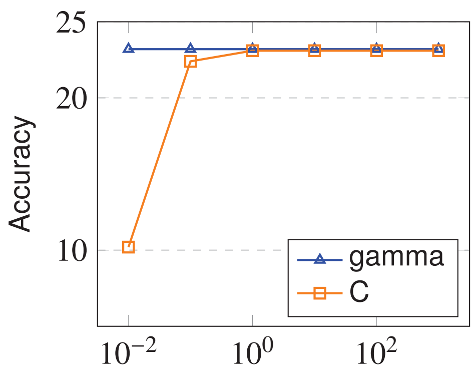


Figure 5.9: Linear SVM accuracy vs algorithm parameters

5.4.2 Design of a Fully-Connected DNN

Using Keras on Tensorflow backend, we designed a fully connected MLP DNN model that holds 2600 inputs and 9 outputs. There are five hidden layers which include 650, 182, 81, 40, and 20 input nodes, respectively. Dropouts are usually used in MLP design to avoid overfitting [84]. A Dropout of 40% is used between layers 3 and 4. The input features are scaled in the range $[0, 1]$. Therefore, we applied the “ReLu” activation function to all layers except for the latest layer that uses the “softmax” activation function. The “softmax” function calculates the probabilities of each target class over all possible target classes. The calculated probabilities will help determine the target class for the given inputs. Therefore, it is the best choice for the output layer. The loss function used to measure the accuracy and validation is the “categorical

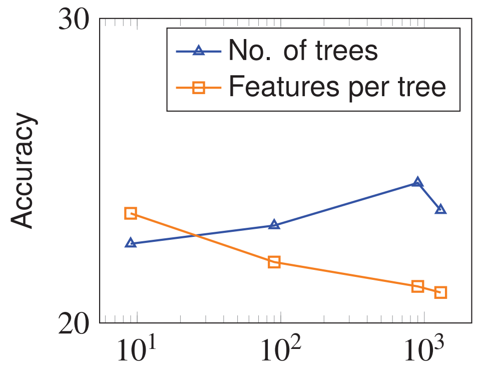


Figure 5.10: Random Forest accuracy on power signal dataset vs parameters

crossentropy". This loss function trains the DNN to output the probability over all the classes of data. Figure 5.11 illustrates our designed fully connected DNN configuration. This model, however, cannot be trained. As Figure 5.12 shows, the validation accuracy does not follow the accuracy per epochs. It stays close to zero while the accuracy converges to one. The validation loss rapidly diverges as well. If we remove the dropout layer between layers 3 and 4, the model will be over-fitted. Consequently, the fully connected MLP method is not suitable for classifying our dataset.

5.4.3 Design of a 1D-CNN

Typically, a 1D-CNN model involves two kinds of layers, one or more Convolution layers; and Multi-layer perceptron layer (MLP). The convolution layer is typically situated just after the input layer. Convolution involves sliding the kernel over the input signal. Convolution is performed in two ways, non-causal (used in typical CNNs) or causal. Non-causal convolution is cross-correlation. That means the output is dependent on future input. Let the input size of length n to convolution layer is represented by x and the kernel of length k is represented by h. Suppose that the kernel window is shifted by s (number of strides) after each convolution operation. Then non-causal convolution y of x and h by stride s is defined as

$$\begin{cases} y(n) = \sum_{i=1}^{k} x(n+i) \cdot h(i), & n = 0 \\ y(n) = \sum_{i=1}^{k} x(n+i+(s-1)) \cdot h(i), & otherwise. \end{cases} \tag{5.2}$$

If padding of length p is used then the number of outputs $o = \lfloor \frac{n+2p-k}{s} \rfloor + 1$. In causal convolution, the output is not dependent on future inputs. the output y is calculated as

$$\begin{cases} y(n) = \sum_{i=0}^{k-1} x(n-i) \cdot h(i), & n = k-1 \\ y(n) = \sum_{i=0}^{k-1} x(n-i+(s-1)) \cdot h(i), & otherwise. \end{cases} \tag{5.3}$$

Pooling is used after the convolution layer to reduce the dimension of the convolution output. It also helps to reduce overfitting. Max-pooling and Average-pooling are two commonly used methods. Max-pooling picks up maximum value in a kernel window and Average- pooling outputs the mean of the kernel window values. The output of convolution layers may have a depth greater than one.

Flatten layer reshapes the output of the convolutional layer to flat that can be fed to the MLP network.

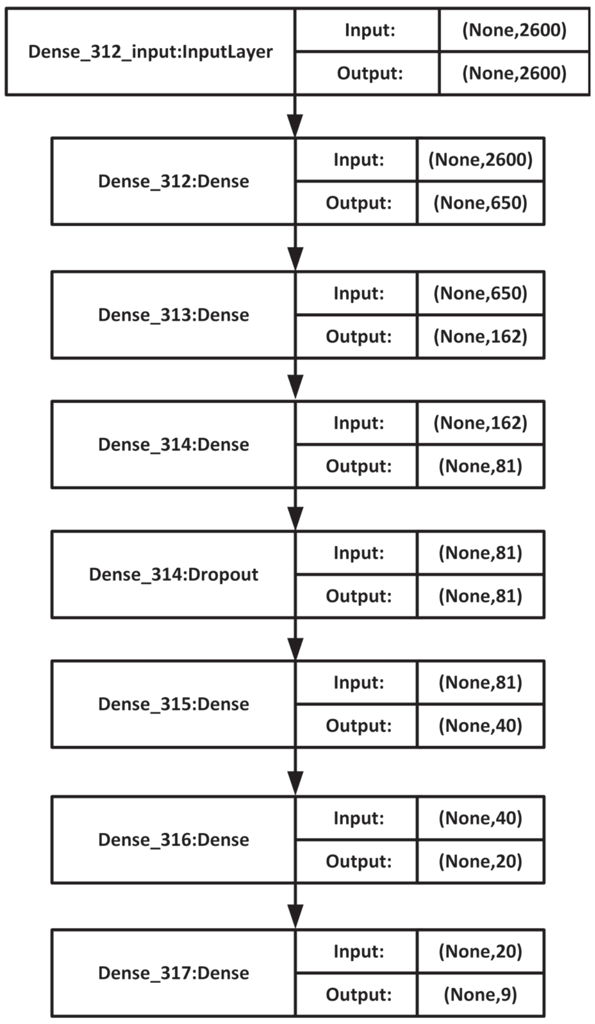


Figure 5.11: Fully connected deep neural network configuration

To set up our 1D-CNN model, we aim to observe power signal changes in two consecutive clock cycles, i.e. changes in every 16 samples in our dataset. We examined two 1D-CNN models. In our first model named Model_1, we designed the first convolutional layer with the kernel of size $k_1 = 16$, and the stride $s_1 = 8$. The inputs to the first convolutional layer are the $n_1 = 2600$ features; we used $f_1 = 2600/8 = 325$ filters for the convolutional layer. Each filter produces an individual convolutional output. The output of the first convolutional layer is $o_1 = \lfloor \frac{2600-16}{8} \rfloor + 1 = 324$. The next convolutional layer uses kernel size $k_2 = 8$, stride $s_2 = 4$, and $f_2 = 110$ filters. The number of outputs is $o_2 = \lfloor \frac{324-8}{4} \rfloor + 1 = 80$. The next layer is the average pooling followed by the Flatten layer that flattens the convolutional layers data to 4400 features. The flattened data are used as inputs of the next MLP (dense) layer. The output of the MLP is equal to the number of data classes which are nine labels. The “ReLu” activation function is used for all convolutional layers. The “softmax” activation is used for the Dense layer (output layer). Similar to the fully connected DNN model, the “categorical crossentropy” loss function is implemented for the Dense layer. In our second model, called Model_2, we used Dataset 2, which consists of $n = 400$ features. Regarding to Figure 5.2, these features include the power samples taken from the multiplication, subtraction, and modular reduction at levels 7 and 8 of ECPD, reading from ROM 1, and multiplication and modular reduction at level 1 of ECPA. Model_2 comprises one convolutional layer with a kernel size of $k = 16$, the

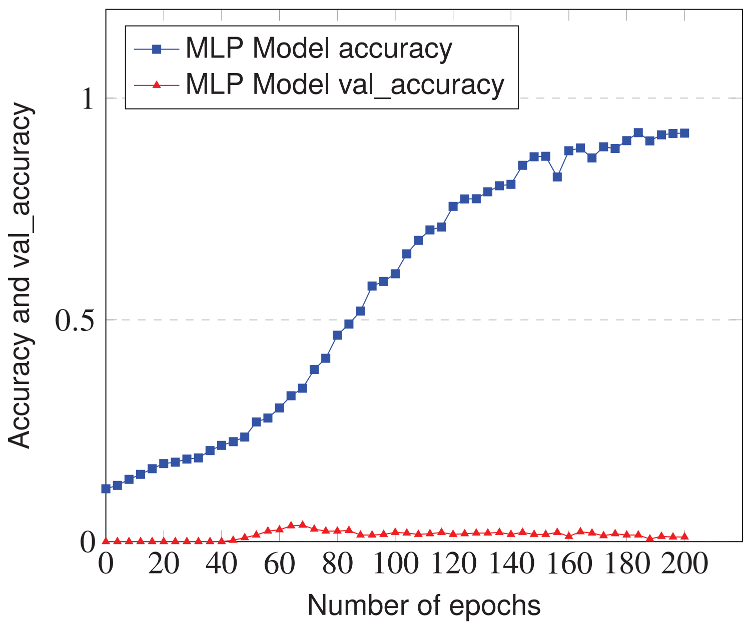


Figure 5.12: Accuracy and accuracy validation for fully connected DNN model.

stride $s = 8$, and $f = 250$ filters. The next layer is a Max-pooling layer, followed by the Flatten layer that converts the convolutional layers data to 7750 features used by the final MLP layer.

Computational complexity of 1D-CNN

To analyse the computational complexity of 1D-CNN, we must compute the total number of operations at each layer (ignoring the sub-sampling that has a negligible computational cost) and then cumulate them to find the overall computational complexity. If the convolutional layer l uses f_l number of filters or feature detectors, kernel size k_l with stride s_l, and $o_l = \lfloor \frac{n_l - k_l}{s_l} \rfloor + 1$ outputs, then the layer l consists of $o_l \times k_l \times f_l$ multiplications and $o_l \times (k_l - 1)$ additions from a single connection. At the first 1D-CNN layer the number of inputs is the number of features, n_0 and at the layer l the number of inputs is $n_l = o_{l-1} \times f_{l-1}$. Ignoring the bias addition, the total number of multiplications and additions in the layer l would be

$$\begin{aligned} mul_l &= f_l o_l k_l, \\ add_l &= o_l(k_l - 1) \end{aligned} \tag{5.4}$$

Therefore, the total number of multiplications and additions ($T(mul)$, and $T(add)$) on a 1D-CNN layer is

$$\begin{aligned} T(mul) &= \sum_{l=1}^{L} o_l k_l f_l, \\ T(add) &= \sum_{l=1}^{L} o_l(k_l - 1). \end{aligned} \tag{5.5}$$

where L is the number of the 1D-CNN layers. The term $T(add)$ is negligible compared to $T(mul)$. Thus the complexity of 1D-CNN model can be considered as

$$O(\sum_{l=1}^{L} o_l k_l f_l). \tag{5.6}$$

The 1D-CNN Model_1 and Model_2 are shown in Figure 5.13 and 5.14, respectively. Both models are fully trainable and can efficiently cluster the power signals into nine categories. Figure 5.16 and Figure 5.17 illustrate training/validation loss and training/validation accuracy per epochs, respectively for both models. The validation accuracy follows training accuracy, revealing that the model is very well trained. The accuracy of Model_1 reaches 99.993% after 80 epochs. Model_2 provides the same accuracy after 250 epochs. However, from (5.6), it can be concluded that Model_2 has much less complexity using one convolutional layer and reduced number of features.

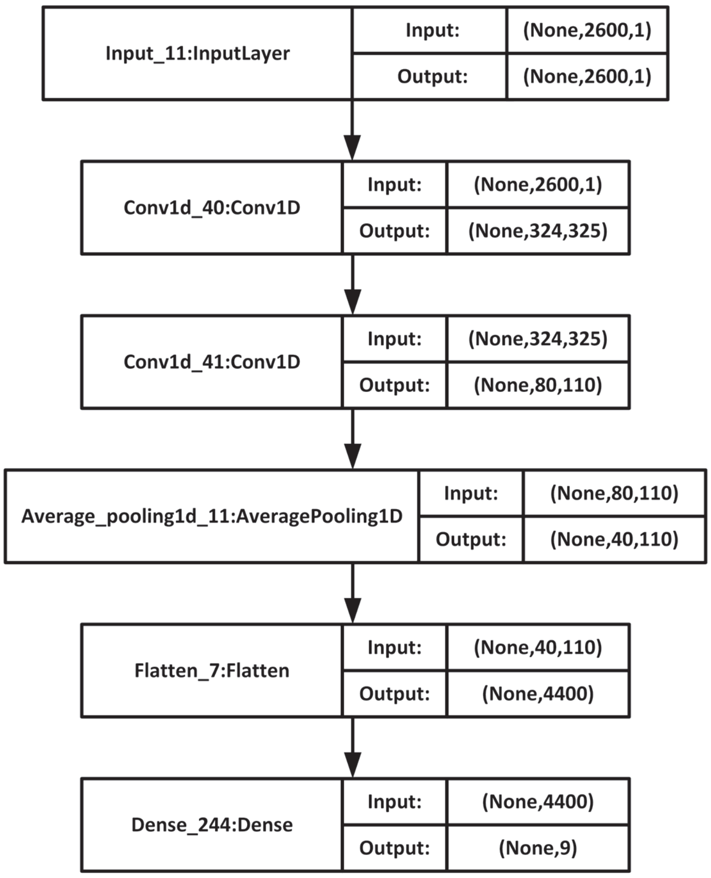


Figure 5.13: 1D-CCN Model_1 configuration

5.5 Applying Countermeasures

The signal power of ROM readings (that is, the PoI) is the only differentiator of the eight ECPD-ECPA operations performed in the core. The readings from the ROM are detectable using a 1D-convolutional neural network. A simple countermeasure could be to hide the patterns of side-channel power data which is correlated with the ROM readings; that is, adding some unpredictable noise to the PoI window that covers the core activity. Practically, this is possible by adding some dummy operators that consume random patterns of power. Since implementation of a true random number generator is costly in hardware, we implemented an extra modular multiplier as a dummy operation which is activated at the end on ECPD. As shown in Figure 5.2, the modular multiplier inputs are taken from RNS channel m_8 to m_1 of registers Z_2 and X_2, respectively. The DSP module of Xilinx series-7 FPGAs has 25×18-bit multipliers

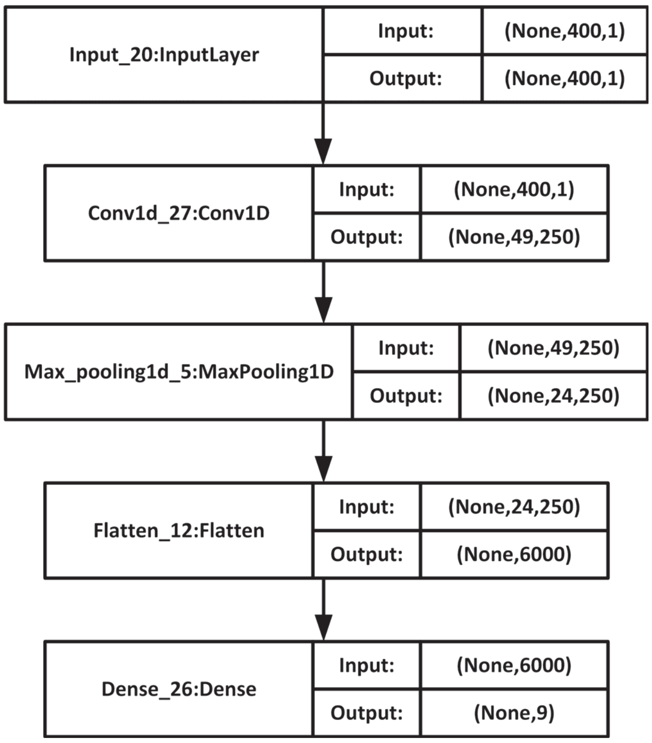
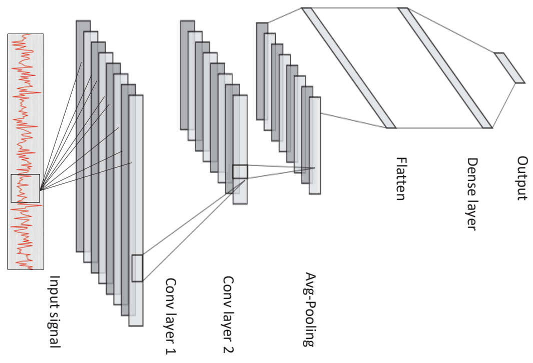
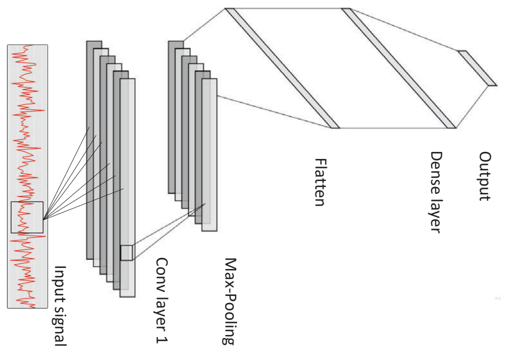


Figure 5.14: 1D-CCN Model_2 configuration



(a) Proposed 1D-CCN Model_1 diagram



(b) Proposed 1D-CCN Model_2 diagram

Figure 5.15: 1D-CNN models

[56]. Our noise generator is built using two 4-bit shift registers and one 25×18-bit multiplier. The inputs of the noise generator are 66-bit X_2 and Z_2 RNS channels. The noise generator is activated for eight clock cycles. At each clock cycle, shift registers shift right the inputs for 4-bits. The least significant 25 bits of X_2 and 18-bits of Z_2 RNS channels are used as inputs of the multiplier. Figure 5.18a shows the block diagram of the noise generator. We used eight blocks of noise generators to produce a sufficient level of non-deterministic power consumption signal. Each noise generator block uses one of eight RNS channels of X_2 and Z_2. Figure 5.18b shows the RNS GLV ECC core architecture with added the noise generator units. At the end of each ECPD operations, the GLV ECC state machine enables noise generators by activating the CLK_EN signal and increases the core power consumption. The output of the noise generators are not used for calculations. Thus, the final result is '*don't care*'. However, they produce a dummy side-channel power signal which covers the ROM reading side-channel

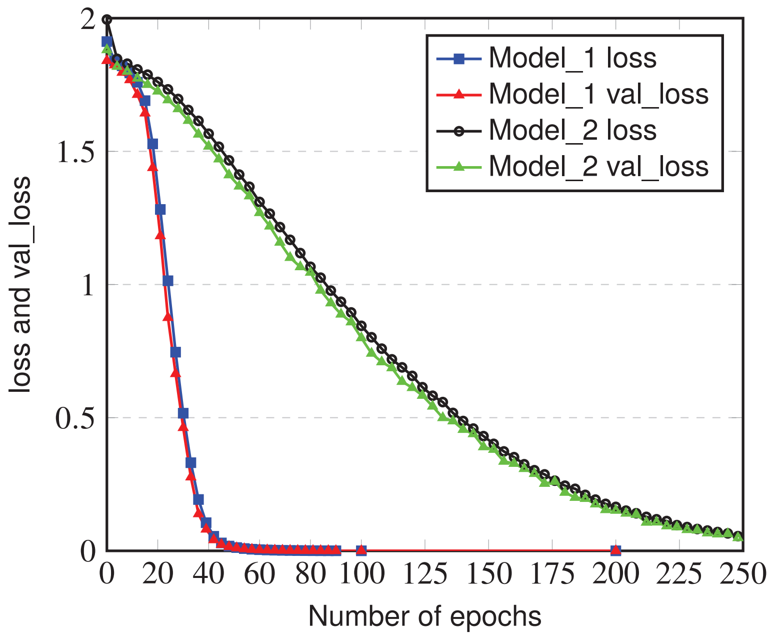


Figure 5.16: Loss and loss validation for 1DCNN Model_1 and Model_2.

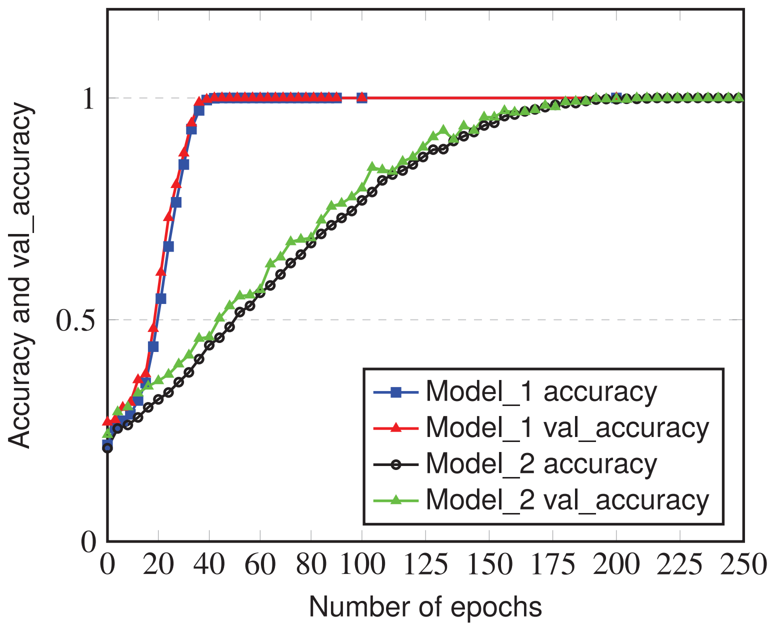
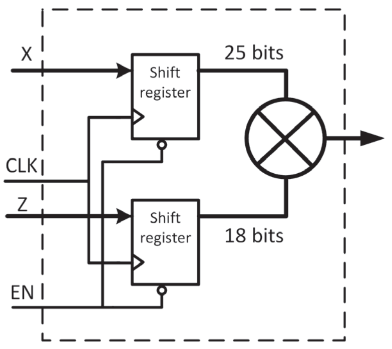
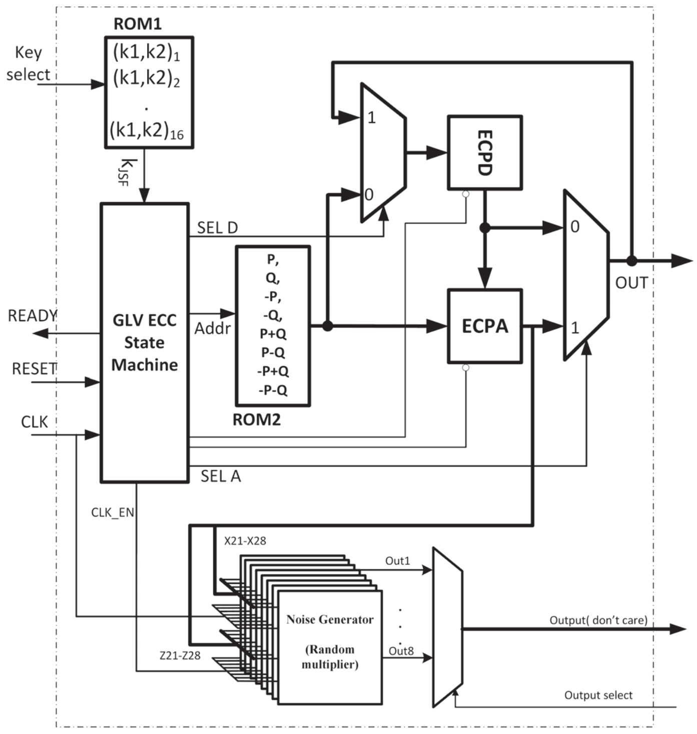


Figure 5.17: Accuracy and accuracy validation for 1DCNN Model_1 and Model_2.

power signal. The side-channel power data of an ECPD-ECPA sequence with the insertion of a dummy multiplier is shown in Figure 5.19. We applied our Model_1 1D-CNN on the new, obtained side-channel data. The model accuracy and validation accuracy graphs are illustrated in Figure 5.20. The accuracy stays around 10%, which means the power signal cannot be categorised by 1D-CNN and our countermeasure is successful.





(b) RNS GLV ECC point multiplication with noise generator.

(a) Noise generator.

Figure 5.18: RNS GLV ECC design immune to side-channel attack.

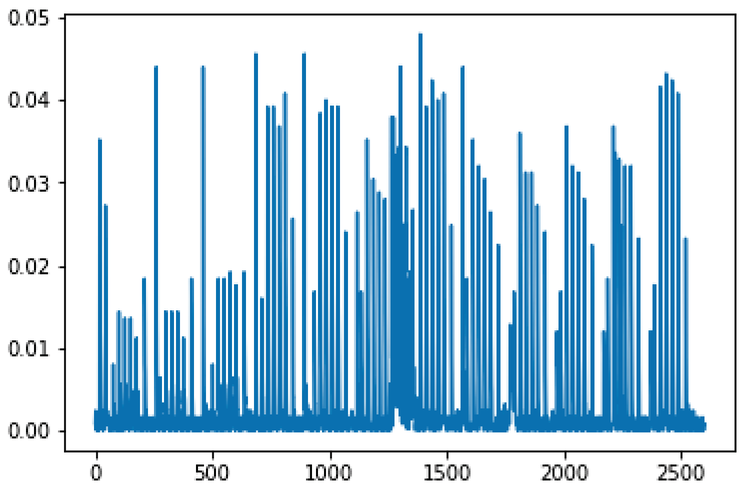


Figure 5.19: ECPD-ECPA power signal with applying a dummy operation

5.6 Conclusion

RNS is considered as a countermeasure against side-channel attacks. We designed an RNS GLV ECC co-processor based on the SOR RNS reduction algorithm. We analysed the side-channel power data of the core performing ECC point multiplication operation. Based on the

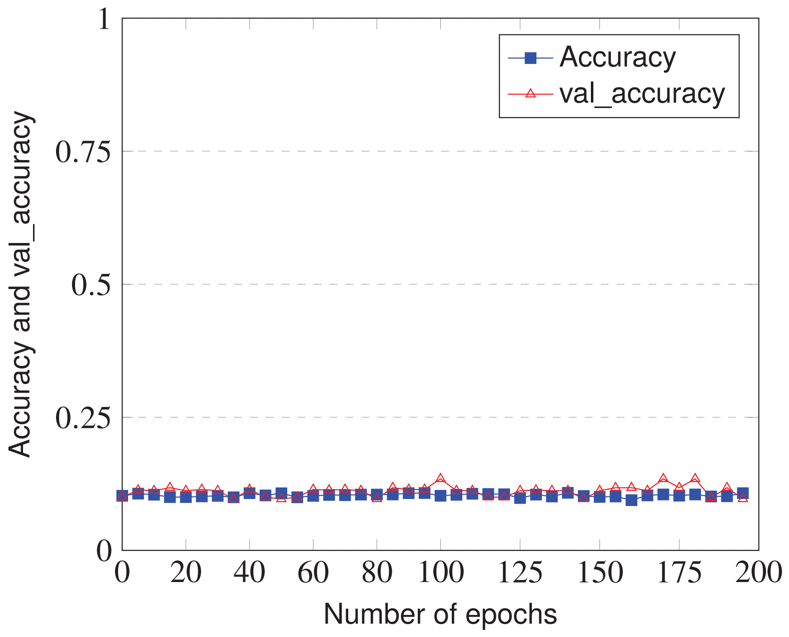


Figure 5.20: 1DCNN Model accuracy/val_accuracy after applying countermeasures

core's internal activity, we categorised the power signals into nine classes. Eight out of nine classes are very similar in terms of the core activity and perform an ECPD–ECPA sequence. Their difference is in readings from the ROM locations, where the pre-computed points are saved. We created a dataset of 4500 samples to train our machine learning algorithms. We used a window of PoI on the side-channel power data that was included the core's power signal when it gains access to its built-in ROM. We tried several machine learning and deep learning methods and found out that a 1D-CNN model can effectively classify the power samples with an accuracy of 99.993%. We narrowed down the PoI window size and used a model with only one 1D-CNN layer. The new model works well and can reach the same accuracy in higher epochs. We concluded that the classifier of side-channel power data is the ROM reading activity of the core. By adding a dummy multiplier to the hardware that activates within the PoI time frame and updating its inputs at every clock cycle, we generated random side-channel power signal that covers the core activities within the PoI window. Applying our 1D-CNN Model_1 to the new side channel power data showed that the accuracy of the model dropped to 10% which means the 1D-CNN cannot classify the operations of the ECC core. Our experimental analysis shows that our proposed cryptographic co-processor is immune to power analysis type of side-channel attacks and it is suitable for hardware implementations. Our work is supporting the idea presented in [118, 119]. The RNS can be used as a countermeasure, however, it is not thoroughly protecting against side-channel attacks. The hardware activity may leak information of the secret key. In our experiment the points of interest which are related to the power consumed by the state machine accessing the ROM can reveal information of the secret key.

6 Conclusion

In this thesis, we introduced a full RNS ECC co-processor. Our target was to reduce the latency of point multiplication operation which is the core security function in ECC.

In Chapter 3, we introduced an improved RNS Montgomery reduction algorithm. Using this algorithm, we achieved to cut the latency in modular reductions by 7% in comparison to the latest works in the literature, while our proposed algorithm implementation on a similar platform utilises fewer hardware resources. We introduced a new RNS modular reduction algorithm based on the sum of residues, which unlike Montgomery modular reduction algorithm returns the precise value of the reduction, not a multiple. Our algorithm — dubbed as SOR— is inherently parallel. Using η RNS multipliers in parallel, this algorithm can calculate the modular reduction in $(\frac{N}{\eta} + 3)$ clock cycles (where N is the number of RNS moduli channels). We introduced two hardware architectures of SOR algorithm. The SOR_1 uses one RNS multiplier and SOR_2 uses two parallel RNS multiplier. The SOR_2 architecture improves the modular reduction speed by more than 20% in comparison to the RNS Montgomery modular reduction implementation. Nevertheless, the area of the SOR algorithm hardware is roughly two times bigger than the area of RNS Montgomery modular reduction implementations, since the algorithm uses the whole range of moduli channels in the computations.

In chapter 4, we proposed RNS ECC point doubling, point addition, and point tripling for three elliptic curves SECP256k1, ED25519, and Brainpool256r1. Using RNS properties and parallel computation with two modular reduction units, we reduced the latency of ECC group operations. Next, we tried Double-and-Add, NAF, NAF$_3$, and DBC point multiplication algorithms to improve the latency of our ECC co-processor. Using the endomorphism property of the curve SECP256k1, we implemented the GLV point algorithm for this curve. The hardware was implemented on VIRTEX 7 and VIRTEX UltraScale + FPGAs to indicate the impact of the chip process technology impacts on our hardware performance as well as the algorithm used.

In Chapter 5, we designed a new RNS GLV ECC co-processor architecture to suit our lab side-channel analysis evaluation board. We used machine-learning and deep-learning algorithms to analyse the side-channel power data we gathered from our designed co-processor. All the studied machine-learning algorithm models and fully connected MLP deep-learning model failed in side-channel power analysis. However, the one-dimensional convolutional neural network model (1D-CNN) was successful in finding a map between the power data and the secret key used by the co-processor. We noticed that the state machine activity in accessing the ROM to read the pre-calculated points could leak information about the secret key. We proposed a countermeasure to mask the side-channel power data associated with the state machine activity when reading from ROM. Our 1D-CNN model was not able to distinguish and categorise the power signals correctly when the proposed countermeasure applied to the RNS GLV co-processor hardware. Our results supported works [118] and [119] that suggested RNS as a countermeasure to side-channel attacks. Nevertheless, RNS is not a thoroughly shield against side-channel attacks, since the core activity may leak information of the secret key.

6.1 Future works and research direction

High performance RNS based elliptic curve cryptographic cores presented in this thesis could be utilised to enhance computing power in a diverse range of applications. The research may be conducted around SECP256k1 core performance within an ECDSA protocol to sign and validate Blockchain transactions and evaluate the overall performance. The Brainpool256r1 is now the recommendation of both IEEE 1609.2b-2019 and European Telecommunications Standards Institute (ETSI) standards for Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) message signing and verification. There is a high demand for high-speed ECC hardware to sign and verify thousands of control and safety messages within a tight time frame. R&D researchers in the automotive industry may be interested in implementing our high-performance RNS based Brainpool256r1 point multiplication hardware within the ITS safety protocols.

IoT has become one of the focused research topics in recent years. The security of data exchange between IoT devices is a significant concern that may breach privacy and theft of sensitive data. Meanwhile, the demand for low-power, light-weight cryptographic hardware suitable to fit in IoT devices is increasing. Generally, RNS computation is costly in terms of power consumption. However, our research outcome could be directive for researchers who are interested in performing trade-offs between different hardware performance parameters such as latency, power consumption, and security.

The study of hardware vulnerabilities against side-channel attacks could be another research direction. A cryptographic algorithm may be considered safe in terms of the underlying maths. However, its implementation can leak data in the form of changes in power consumption, electromagnetic radiation or timing. The side-channel data leakage is a critical security thread for any cryptographic hardware implementation. Some researchers may focus on new methods of side-channel data analysis to reveal the sources of data leakage when others may research on countermeasures. Design of countermeasures in the context of RNS arithmetic, for instance by randomising RNS bases, can be studied in future works. Our research showed that 1D-CNN effectively reveals the correlation of power signal changes and internal operation per clock cycles. Research on time-based deep learning algorithms such as Recurrent Neural Networks (RNN) and Generative Adversarial Networks (GAN) to analyse side-channel data and design of the effective and low-cost countermeasures will remain as future works.

A

Appendix A

A.1 Maple program for testing proposed RNS Montgomery reduction algorithm

```
m(1) := 2^66-1; m(2) := 2^66-2^2-1; m(3) := 2^66-2^3-1; m(4) := 2^66-2^4-1;
m(5) := 2^66-2^5-1; m(6) := 2^66-2^6-1; m(7) := 2^66-2^8-1; m(8) := 2^66-2^9-1;
k(1) := 2^66-1; k(2) := 2^66-2^2-1; k(3) := 2^66-2^3-1; k(4) := 2^66-2^4-1;
q(1) := 2^66-2^5-1; q(2) := 2^66-2^6-1; q(3) := 2^66-2^8-1; q(4) := 2^66-2^9-1;
p := 2^256-2^32-2^9-2^8-2^7-2^6-2^4-1;
A := 2^260-2^40-123; B := 2^256-135;
K := 1;
for i to 4 do K := K*k(i) end do; for i to 4 do KI(i) := K/k(i) end do;
for i to 4 do Kinv(i) := modp(1/KI(i), k(i)) end do;
Q := 1;
for i to 4 do Q := Q*q(i) end do; for i to 4 do QI(i) := Q/q(i) end do;
for i to 4 do Qinv(i) := modp(1/QI(i), q(i)) end do;
for i to 4 do
for j to 4 do (KIJ(i))(j) := modp(KI(i), q(j)) end do:
end do;
for i to 4 do
for j to 4 do (QIJ(i))(j) := modp(QI(i), k(j)) end do:
end do;
for i to 4 do PNQ(i) := modp((-p)^(-1), q(i)) end do;
for i to 4 do PK(i) := modp(p, k(i)) end do;
for i to 4 do QNK(i) := modp(1/Q, k(i)) end do;
for i to 4 do IQNK(i) := modp(Qinv(i)*Kinv(i), k(i)) end do;
for i to 4 do IPNQ(i) := modp(Qinv(i)*PNQ(i), q(i)) end do;
for i to 4 do PNK(i) := modp(1/p, k(i)) end do;
for i to 4 do
ABK(i) := modp(A*B, k(i)); ABQ(i) := modp(A*B, q(i))
end do;
for i to 4 do GammaQ(i) := modp(ABQ(i)*IPNQ(i), q(i)) end do;
for i to 4 do ABK(i) := modp(ABK(i)/Q, k(i)) end do;
for i to 4 do
for j to 4 do
```

```
(QK(i))(j):= modp(GammaQ(i)*QNK(j)*PK(j)*(QIJ(i))(j), k(j)):
end do:
end do;
for i to 4 do Y(i) := 0 end do;
for i to 4 do
for j to 4 do Y(i) := modp(Y(i)+(QK(j))(i), k(i)) end do:
end do;
Al2 := 0; for i to 4 do Al2 := Al2+GammaQ(i)/q(i) end do; alpha := floor(Al2);
for i to 4 do alphaQ(i) := modp(-alpha*p, k(i)) end do;
for i to 4 do BEQK(i) := modp(Y(i)+alphaQ(i)+ABK(i), k(i)) end do;
for i to 4 do GammaK(i) := modp(BEQK(i)*Kinv(i), k(i)) end do;
for i to 4 do
for j to 4 do (KQ(i))(j) := modp(GammaK(i)*(KIJ(i))(j), q(j)): end do:
end do;
for i to 4 do Z(i) := 0 end do;
for i to 4 do for j to 4 do Z(i) := modp(Z(i)+(KQ(j))(i), q(i)) end do end do;
Al3 := 0; for i to 4 do Al3 := Al3+GammaK(i)/k(i) end do; alpha := floor(Al3);
for i to 4 do alphaK(i) := modp(-alpha*K, q(i)) end do;
for i to 4 do BEKQ(i) := modp(Z(i)+alphaK(i), q(i)) end do;
for i to 4 do printf("␣%X␣,␣␣", BEQK(i)) end do;
 166C6008EF3D82EC6 ,   6F2C59E44FA4D738 ,   2E0098061ABF2F9F1 ,   358A2766BA909CE5D ,
for i to 4 do printf("␣%X␣,␣␣", BEKQ(i)) end do;
 33A6D1DF89DC94C3 ,   203B79F5D8BEDB58C ,   18638AB5E61110F9F ,   16B0EBC20307A381B ,
X := modp(A*B/Q, p);
X := 25885064547426835970113115578390014457565065405408988046730615365211366033878
for i to 8 do printf("%X␣␣,␣", modp(X, m(i))) end do;
166C6008EF3D82EC6  , 6F2C59E44FA4D738  , 2E0098061ABF2F9F1  , 358A2766BA909CE5D
,
33A6D1DF89DC94C3  , 203B79F5D8BEDB58C  , 18638AB5E61110F9F  , 16B0EBC20307A381B
,
```

A.2 Maple program for testing sum of residues reduction algorithm

```
with(NumberTheory):
A := 2^260-2^40-123: B := 2^256-135: X := A*B:
p := 2^256-2^32-2^9-2^8-2^7-2^6-2^4-1:
m(1) := 2^66-1: m(2) := 2^66-2^2-1: m(3) := 2^66-2^3-1: m(4) := 2^66-2^4-1:
m(5) := 2^66-2^5-1: m(6) := 2^66-2^6-1: m(7) := 2^66-2^8-1: m(8) := 2^66-2^9-1:
M := 1: for i to 8 do M := M*m(i) end do:
for i to 8 do MI(i) := M/m(i) end do:
for i to 8 do MIinv(i) := modp(1/MI(i), m(i)) end do:
for i to 8 do Gamma(i) := modp((modp(X, m(i)))*MIinv(i), m(i)) end do:
Al2 := 0: for i to 8 do Al2 := Al2+Gamma(i)/m(i) end do:
alpha := floor(Al2): alpha:
a := 0: for i to 8 do a := a+floor(Gamma(i)/2^58) end do:
a2 := floor((a+2^4)/2^8):
KZS := 0: for i to 8 do KZS := KZS+Gamma(i)*(modp(MI(i), p))/p end do:
KZS := trunc(KZS):
KZ := 0: for i to 8 do KZ := KZ+Gamma(i)*floor((modp(MI(i), p))/2^184) end do:
Kappa := floor(KZ/2^72):
for i to 8 do Ki(i) := modp(Kappa, m(i)) end do:
for i to 8 do pi(i) := modp(-p, m(i)) end do;
for i to 8 do Kp(i) := modp(Ki(i)*pi(i), m(i)) end do:
for i to 8 do
for j to 8 do (MI_M_mj(i))(j) := modp(modp(MI(i), p), m(j)) end do:
```

```
end do:
for i to 8 do
for j to 8 do (Y(i))(j) := modp(Gamma(i)*(MI_M_mj(i))(j), m(j)) end do:
end do:
for i to 8 do sigma(i) := 0 end do;
for i to 8 do
for j to 8 do sigma(i) := modp(sigma(i)+(Y(j))(i), m(i)) end do:
end do:
for i to 8 do alphaM(i) := modp(modp(-alpha*M, p), m(i)) end do:
for i to 8 do Z(i) := modp(sigma(i)+alphaM(i), m(i)) end do:
for i to 8 do z(i) := modp(Z(i)+Kp(i), m(i)) end do: printf("\n");
for i to 8 do printf("␣%X␣,␣␣", z(i)) end do:
```

A.3 Maple program for DBC Tree algorithm

```
with(StringTools):
f  := fopen( "D:\\C␣PROJECTS\\RND-256-BIT.txt", READ ):
fw := fopen( "D:\\C␣PROJECTS\\Mapleout.txt", WRITE):
samples := 200:
AVL:=0:
for h from 1 to samples do
b:= readline(f):
a:=convert(Chop(b),decimal,hex):
k:=0:
i:=0:
j:=0:
ri:=0:
rj:=0:
li:=0:
lj:=0:
if (modp(a,2)=0 or modp(a,3)=0) then

while (modp(a,2)=0) do
        a:= a/2:
        i:= i+1:
end do:

while (modp(a,3)=0) do
        a:= a/3:
        j:=j+1:
end do:
u(k) := i:
v(k) := j:
w(k) := 0:
k:=k+1:
end if:

while (a>1) do

ri:=0:
rj:=0:
li:=0:
lj:=0:

la:= a-1:
ra:= a+1:
while (modp(la,2)=0) do
```

```
        la:= la/2:
        li:=li+1:
end do:

while (modp(la,3)=0) do
        la:= la/3:
        lj:=lj+1:
end do:

while (modp(ra,2)=0) do
        ra:= ra/2:
        ri:=    ri+1:
end do:
while (modp(ra,3)=0) do
        ra:= ra/3:
        rj:=rj+1:
end do:

if ra >= la then
        a:= la:
        u(k):=  li:
        v(k):=  lj:
        w(k):= +1:
else
        a:= ra:
        u(k):=  ri:
     v(k):=  rj:
     w(k):=  -1:
end if:
k:= k+1:
end do:

for l from 0 to k-2 do
printf("3^%d*2^%d(", v(l), u(l));
end do:
printf("3^%d*2^%d", v(k-1),u(k-1));
for l from k-1 to 1 by -1 do
printf("%-+d)", w(l));
end do:
if (w(0) <> 0) then
printf("%-+d", w(0));
end if:
printf("\n,␣Length:=%d␣\n", k);

printf( "B(%2d):=Matrix([",h);
fprintf(fw,"B(%2d):=Matrix([", h);
for l from k-1 to 0 by -1 do
        for xx from 1 to u(l) do
                printf("%d,",2);
                fprintf(fw,"%d,",2);
     end do:
     for xx from 1 to v(l) do
        printf("%d,",3);
        fprintf(fw,"%d,",3);
     end do:
     if w(l) = 1 then
        printf("%d,",1);
        fprintf(fw,"%-d,",1);
     elif w(l) = -1 then
        printf("%-d,",-1);
        fprintf(fw,"%-d,",-1);
     end if:
end do:
printf("]):\n"):
fprintf(fw,"]):\n"):
AVL:= AVL+k:
printf("\n\n");
end do:
AVL:= evalf(AVL/samples);
fclose(f):
fclose(fw):
```

A.4 L-T DBC algorithm in C++

```
/* functions written by:  Cristobal Leiva and Nicolas Theriault
main body written by Mohamadali Mehrabi.
 */
#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <bitset>
#include <iostream>
#ifndef BITS
    #define BITS 256
#endif
#include <iostream>
#include <conio.h>
#include <fstream>
#include <algorithm>
using namespace std;
static const int64_t max_size = BITS+4;
/*
 * Weights of positive (P_weight) and negative (N_weight) chains.
 * We just need two rows of information for previous and current chains.
int64_t P_weight[2][max_size], N_weight[2][max_size];
/*
 * 'Movements array' for storing information for every chain.
 * Every term needs 4 bits of information:
 * - 1 bit to know whether we did a doubling (0) or tripling (1) to get here
 *   (horizontal or vertical step).
 * - 1 bit to know whether the previous chain was positive (0) or negative (1).
 * - 2 bits to know if we got here by doing nothing (00), adding a term (01)
 *  or substracting a term (11).
 *  For example, if we got to some chain by making a vertical step
   (tripling) from a negative chain and we added a negative term,
   that means $n_{i,j} = \overline{\mathscr{C}}_{i,j-1} - 2^{i}3^{j-1}$.
   In this code, this will be denoted as (V, -, -1).
 * int8_t provides 8 bits, so we use the first half of bits for
 storing information of positive chains and last 4 bits for storing
 information of negative chains.
 * Example: We zero first half of bits and then store a movement of the type
   which translates to setting bits 0011:
```

```
 * T[j][i+1] &= ~(15 << 0);
 * T[j][i+1] |= (3 << 0);
 * Example: We zero second half of bits and then store a movement of the type
   (V, -, -1) which translates to settings bits 1111:
 * T[j+1][i] &= ~(15 << 4);
 * T[j+1][i] |= (15 << 4);
 * */
int8_t T[max_size][max_size];
typedef struct {
    int64_t weight;
    int64_t i;
    int64_t j;
} chain_t;
typedef chain_t* chainptr_t;

typedef struct {
    std::bitset<max_size> num;
    int64_t msb;
    bool zero;
} bigint_t;

typedef bigint_t* bigintptr_t;

/* Convert big integers from string hexadecimal representation
a bitset */
void str_to_bits(char *orig, bigintptr_t dest)
{
    uint8_t digit;
    uint64_t bitcount = 0;
    std::bitset<max_size> mask;
    for(int64_t i = strlen(orig)-1; i >= 0; i--)
    {
        digit = (orig[i] > '9') ? (orig[i] &~ 0x20)-'A'+10: (orig[i]-'0');
        mask = std::bitset<max_size>(digit);
        dest->num |= (mask << bitcount);
        bitcount += 4;
    }
    dest->zero = (bitcount > 0) ? false: true;
    dest->msb = bitcount;
}

/* Manually divide by 3 by looking at bits and keeping track of carries */
void divide_by_3(bigintptr_t orig, bigintptr_t dest)
{
    uint8_t carry = 0;
    orig->zero = true;
    dest->zero = true;
    dest->msb = 0;
    dest->num.reset();
    for(int64_t i = orig->msb; i >= 0; i--)
    {
        if(orig->num.test(i))
        {
            if(carry == 1)
            {
                dest->num.set(i);
                carry--;
                dest->zero = false;
                if(i > dest->msb) dest->msb = i+1;
            }
            else if(carry == 2)
            {
                dest->num.set(i);
                dest->zero = false;
                if(i > dest->msb) dest->msb = i+1;
            }
            else
                carry++;
            orig->zero = false;
        }
        else
        {
            if(carry == 1)
                carry++;
            else if(carry == 2)
            {
                dest->num.set(i);
                carry--;
                dest->zero = false;
                if(i > dest->msb) dest->msb = i+1;
            }
        }
    }
}

void optimal_chain(bigint_t a, chainptr_t shortest)
{
    bigint_t b ;
    int64_t i, j, size, cont;
    int8_t curr, next, aux;
    /* Initialization */
    for(i = 0; i < max_size; i++)
        P_weight[0][i] = N_weight[0][i] = max_size;
    shortest->weight = max_size;
    P_weight[0][0] = 0; /* base case */
    j = 0;
    curr = 0;
    next = 1;
    while(!a.zero)
    {
        divide_by_3(&a, &b);
        cont = 0;
        size = a.msb;
        P_weight[next][size+1] = N_weight[next][size+1] = max_size;
        P_weight[next][size+2] = N_weight[next][size+2] = max_size;
        for(i = 0; i <= size; i++)
        {
 /* We don't need to check all the cases if weights of both
    positive and negative chains are equal or greater than
    shortest chain found so far */
        if(P_weight[curr][i] >= shortest->weight &&
        N_weight[curr][i] >= shortest->weight)
        {
            P_weight[next][i] = N_weight[next][i] = max_size;
            cont++;
        }
        else
        {
            /* Horizontal steps */
            if(a.num.test(i))
            {
                if(N_weight[curr][i] < N_weight[curr][i+1])
                /* (H, -, 0) */
                {
                    N_weight[curr][i+1] = N_weight[curr][i];
                    T[j][i+1] &= 15;
                    T[j][i+1] |= 64; /* 0100 */
                }
                if(P_weight[curr][i]+1 < P_weight[curr][i+1])
                /* (H, +, +1) */
                {
                    P_weight[curr][i+1] = P_weight[curr][i]+1;
                    T[j][i+1] &= 240;
                    T[j][i+1] |= 1; /* 0001 */
                }
                if(P_weight[curr][i]+1 < N_weight[curr][i+1])
                /* (H, +, -1) */
                {
                    N_weight[curr][i+1] = P_weight[curr][i]+1;
                    T[j][i+1] &= 15;
                    T[j][i+1] |= 48; /* 0011 */
                }
            }
            else /* bit == 0 */
            {
                if(P_weight[curr][i] < P_weight[curr][i+1]) /* (H, +, 0) */
                {
                    P_weight[curr][i+1] = P_weight[curr][i];
                    T[j][i+1] &= 240;
                }
                if(N_weight[curr][i]+1 < N_weight[curr][i+1]) /* (H, -, -1) */
                {
                    N_weight[curr][i+1] = N_weight[curr][i]+1;
                    T[j][i+1] &= 15;
                    T[j][i+1] |= 112; /* 0111 */
                }
                if(N_weight[curr][i]+1 < P_weight[curr][i+1]) /* (H, -, +1) */
                {
                    P_weight[curr][i+1] = N_weight[curr][i]+1;
                    T[j][i+1] &= 240;
                    T[j][i+1] |= 5; /* 0101 */
                }
            }
            /* Vertical steps */
            if(!a.zero)
            {
                if(a.num.test(i) ^ b.num.test(i))
                {
                    P_weight[next][i] = P_weight[curr][i]+1;
                    N_weight[next][i] = N_weight[curr][i]+1;
                    T[j+1][i] = 249; /* 1111 and 1001 */
                }
                else
                {
                    if(a.num.test(i) ^ a.num.test(i+1) ^ b.num.test(i+1))
                    {
                        P_weight[next][i] = max_size;
                        N_weight[next][i] = P_weight[curr][i]+1;
                        T[j+1][i] = 176; /* 1011 and 0000 */
                        if(N_weight[curr][i] < N_weight[next][i])
                        /* (V, -, 0) */
                        {
                            N_weight[next][i] = N_weight[curr][i];
                            T[j+1][i] &= 15;
                            T[j+1][i] |= 192; /* 1100 */
                        }
                    }
                    else
                    {
                     N_weight[next][i] = max_size;
                     P_weight[next][i] = P_weight[curr][i];
                     T[j+1][i] = 8; /* 0000 and 1000 */
                     if(N_weight[curr][i]+1 < P_weight[next][i])
                     /* (V, -, +1) */
                      {
                        P_weight[next][i] = N_weight[curr][i]+1;
                        T[j+1][i] &= 240;
                        T[j+1][i] |= 13; /* 1101 */
                        }
                    }
                }
            }
        }
    }
    /* Check if this iteration produced a chain shorter
    than the shortest so far */
    if(P_weight[curr][size+1] < shortest->weight)
    {
        shortest->weight = P_weight[curr][size+1];
        shortest->i = size+1;
        shortest->j = j;
    }
    if(P_weight[curr][size+2] < shortest->weight)
    {
        shortest->weight = P_weight[curr][size+2];
        shortest->i = size+2;
        shortest->j = j;
    }
    if(a.zero) break;
    size = b.msb;
    if(P_weight[next][size+1] < shortest->weight)
    {
        shortest->weight = P_weight[next][size+1];
        shortest->i = size+1;
        shortest->j = j+1;
    }
    if(P_weight[next][size+2] < shortest->weight)
    {
        shortest->weight = P_weight[next][size+2];
        shortest->i = size+2;
        shortest->j = j+1;
    }
    if(cont >= a.msb) break;
    a = b;
    j++;
    aux = curr;
    curr = next;
    next = aux;
}
}
/* Backtrack from the information of the shortest chain
   and the information array T */
void print_chain(chainptr_t chain)
{
    int64_t i = chain->i;
    int64_t j = chain->j;
    int8_t base = 0;
    bool type = T[j][i] & 8;
    bool sign = T[j][i] & 4;
    bool change_sign = T[j][i] & 2;
    bool change_value = T[j][i] & 1;
    while(chain->weight > 0)
    {
        (type) ? (j--): (i--);
        if(change_value)
        {
            (change_sign) ? (printf("␣-␣")): (printf("␣+␣"));
            cout << "2^" << i << "*3^" << j;
            chain->weight--;
        }
        (sign) ? (base = 4): (base = 0);
        type = T[j][i] & (1 << (base+3));
        sign = T[j][i] & (1 << (base+2));
        change_sign = T[j][i] & (1 << (base+1));
        change_value = T[j][i] & (1 << base);
    }
    printf("\n");
}
void dbc(chainptr_t chain)
{
    int64_t i = chain->i;
    int64_t j = chain->j;
    int64_t w = chain->weight;
    int8_t base = 0;
    int n;
    char s[w];
    int64_t k,l;
    int64_t  a[w],b[w];
    bool type = T[j][i] & 8;
    bool sign = T[j][i] & 4;
    bool change_sign = T[j][i] & 2;
    bool change_value = T[j][i] & 1;
    a[0]=0;
    b[0]=0;

    l= chain->weight;
     printf("%d␣",l); printf("\n\n");

    while(chain->weight > 0)
    {
        (type) ? (j--): (i--);
        if(change_value)
        {
            (change_sign) ? (s[chain->weight]='-'): (s[chain->weight]='+');
            a[chain->weight]= i;
                        b[chain->weight]= j;
            chain->weight--;
        }
        (sign) ? (base = 4): (base = 0);
        type = T[j][i] & (1 << (base+3));
        sign = T[j][i] & (1 << (base+2));
        change_sign = T[j][i] & (1 << (base+1));
        change_value = T[j][i] & (1 << base);
   }
for(k= l; k>= 0; k--)
        printf("(%d,%d)␣", a[k],b[k]);

for(k= l-1; k> 0; k--)
        printf("(%c)␣", s[k]);

printf("\n\n");

for (k = 0 ; k < l-1; k++)
{
 cout << "2^" << a[k+1]-a[k] << "*" << "3^" << b[k+1]-b[k] << "(" ;
}
cout << "2^" << a[l]-a[l-1] << "*" << "3^" << b[l]-b[l-1];
for (k = l-1 ; k >0  ; k--)
{
cout << s[k] << "1" << ")";
}
printf("\n\n");

/*for (k = l-1 ; k >0 ; k--)
{
 cout << "(" << a[k+1]-a[k] << ")" << "(" << b[k+1]-b[k] << ")" << s[k] ;
}
cout << "(" << a[l]-a[l-1] << ")" << "(" << b[l]-b[l-1]<< ")"; */

for (k = l-1 ; k >0 ; k--)
{
cout << "(" << a[k+1]-a[k] <<"," << b[k+1]-b[k] << "," << s[k]<<")" << ",␣";
}
cout << "(" << a[1] <<","<< b[1]<< ")";
printf("\n\n");

/*for (k = l-1 ; k >0 ; k--)
{
for(n=1; n<=a[k+1]-a[k] ; n++ ) printf("[2,0] ");
for(n=1; n<=b[k+1]-b[k] ; n++ ) printf("[3,0] ");
if (s[k]= '-')  printf("[0,1] ");
else printf("[0,-1] ");
}
for(n=1; n<=a[l]-a[l-1] ; n++ ) printf("[2,0] ");
for(n=1; n<=b[l]-b[l-1] ; n++ ) printf("[3,0] "); */
printf("\n\n");
printf("[␣");
for (k = l-1 ; k >0 ; k--)
```

```
{
for(n=1; n<=(a[k+1]-a[k]); n++ ) printf("2,␣");
for(n=1; n<=b[k+1]-b[k] ; n++ ) printf("3,␣");
if (s[k]=='+')  printf("1,␣");
if (s[k]=='-') printf("-1,␣");
}
for(n=1; n<=a[1] ; n++ ) printf("2,␣");
for(n=1; n<=b[1] ; n++ ) printf("3,␣");
printf("␣]");
printf("\n\n");
}
void dbchain(chainptr_t chain)
{
        ofstream fout;
    fout.open("DBCOUT255.dat", ios::app);

    int64_t i = chain->i;
    int64_t j = chain->j;
    int64_t w = chain->weight;
    int8_t base = 0;
    int n;
    char s[w];
    int64_t k,l;
    int64_t  a[w],b[w];
    bool type = T[j][i] & 8;
    bool sign = T[j][i] & 4;
    bool change_sign = T[j][i] & 2;
    bool change_value = T[j][i] & 1;
    a[0]=0;
    b[0]=0;
    l= chain->weight;
     printf("%d␣",l); printf("\n\n");

    while(chain->weight > 0)
    {
        (type) ? (j--): (i--);
        if(change_value)
        {
            (change_sign) ? (s[chain->weight]='-'): (s[chain->weight]='+');
            a[chain->weight]= i;
                        b[chain->weight]= j;
            chain->weight--;
        }
        (sign) ? (base = 4): (base = 0);
        type = T[j][i] & (1 << (base+3));
        sign = T[j][i] & (1 << (base+2));
        change_sign = T[j][i] & (1 << (base+1));
        change_value = T[j][i] & (1 << base);
   }
    fout << "[␣" ;
    for (k = l-1 ; k >0 ; k--)
      {
       for(n=1; n<=(a[k+1]-a[k]); n++ ) fout << "2,␣";
       for(n=1; n<=b[k+1]-b[k] ; n++ )  fout << "3,␣";
       if (s[k]=='+')  fout << "1,␣";
       if (s[k]=='-')  fout << "-1,␣";
      }
    for(n=1; n<=a[1] ; n++ ) fout << "2,␣";
    for(n=1; n<=b[1] ; n++ ) fout << "3,␣";
    fout << "␣]" << "\n";
    fout.close();
}
int main()
{
char *scalar;
int i,num_characters;
string line;
float average, iav,jav;
average = 0;
iav=0;
jav=0;
 ifstream inFile;
 inFile.open("D:\\C␣PROJECTS\\RND-256-BIT.txt");
// ofstream outFile;
// outFile.open("DBC.dat", ios::out);

     if (inFile.is_open())
        {
         // while ( !inFile.eof())
         for (i=1;i<=3 ;i++)
          {
           getline(inFile, line);
           line.erase(64);
                   scalar= &line[0];
           num_characters ++;
           cout<< scalar << endl;
           bigint_t n;
           str_to_bits(scalar , &n);
           chain_t shortest;
           optimal_chain(n, &shortest);
           average += shortest.weight;
           iav += shortest.i;
           jav += shortest.j;
           cout << "Minimal␣Hamming␣Weight:␣" << shortest.weight <<

           std::endl << "Example␣of␣an␣actual␣chain:" ;
           print_chain(&shortest); cout << average<< endl;
           }
       }
     else
        cout << "File␣cannot␣be␣opened" << endl ;
inFile.close();
cout << endl;
average = average/1000;
iav= iav/1000;
jav= jav/1000;
cout << "The␣average␣length␣of␣DBC␣is␣:␣" << average << endl;
cout << "The␣average␣2's␣power␣of␣DBC␣is␣:␣" << iav << endl;
cout << "The␣average␣3's␣power␣of␣DBC␣is␣:␣" << jav << endl;
getch();
return 0;
}
```

A.5 Python code for creating training dataset from the raw side-channel power data

```python
"""
Created on Tue Dec 10 13:45:58 2019
@author: Ali Mehrabi
Creating training dataset from raw side-channel power data
"""
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import time as t
######################################################################
start_time = t.time()
FILE         = "C:\\Users\\Ali\\Downloads\\PA"
EXT          = ".csv"
FILE_DS      = "D:\\Dataset2.csv"
RLBLFILE     = "D:\\Y2.csv"
U            = 199104              #  24888*int(SP)
FRAME        = 4384
DFRAME       = 1310                #
n_samples    = 500
n_slice      = 10
n_rows       = int(n_samples/ n_slice)
######################################################################
############## Read 40 rows , each row takes 10 D/A data ############
for k in range(0,8):
    D  = np.loadtxt(FILE+
                str(k)+EXT, delimiter=',',skiprows=1, max_rows=n_rows)
    D  = D[:, U:U+13*FRAME]
    for j in range(0,n_rows):
        DF = D[j,FRAME: 2*FRAME].reshape(1,FRAME)
        for i in range (2,n_slice+1):
            DS = D[j,i*FRAME:(i+1)*FRAME].reshape(1,FRAME)
            DF = np.concatenate((DF,DS), axis=0)
        DF = pd.DataFrame(DF)
        DF.to_csv(FILE_DS, mode='a', header= False, index= False)
#######################################################################
Z  = np.zeros(FRAME-DFRAME).reshape(1,FRAME-DFRAME)
D  = np.loadtxt(FILE+ str(8)+EXT, delimiter=',',skiprows=1, max_rows=n_rows)
D  = D[:, U:U+13*DFRAME]
for j in range(0,n_rows):
    DF = D[j,DFRAME: 2*DFRAME].reshape(1,DFRAME)
    DF = np.concatenate((DF, Z), axis=1)
    for i in range (2,n_slice+1):
        DS = D[j,i*DFRAME:(i+1)*DFRAME].reshape(1,DFRAME)
        DS = np.concatenate((DS, Z), axis=1)
        DF = np.concatenate((DF,DS), axis=0)
    DF = pd.DataFrame(DF)
    DF.to_csv(FILE_DS, mode='a', header= False, index= False)
```

```
####################### Y_test generation ###############################
y=np.array([])
yr = np.ones(n_samples)
for j in range(1,9):
    y=np.append(y,j*yr,axis=0)
y= np.append(y,np.zeros(n_samples),axis=0)
y = pd.DataFrame(y)
y.to_csv(RLBLFILE, index=False, header = False)
########################################################################
end_time = t.time()
run_time = end_time-start_time
print("␣Total␣time:␣%.2f␣sec" %run_time)
```

A.6 Python code for fully connected deep-learning model

```
"""
Created on Sun Dec  8 22:26:09 2019
@author: Ali Mehrabi
Fully connected model
in rev 4 the dataset increased to 500 rows for each D/A probablity
"""

import numpy as np
import matplotlib.pyplot as plt
from   sklearn.model_selection import train_test_split
from   sklearn          import preprocessing  as pp
from   keras.utils      import  np_utils, plot_model
from   keras.models     import  Sequential
from   keras.layers     import  Dense, Dropout
from   keras.losses     import  categorical_crossentropy,
mean_squared_error, categorical_hinge,sparse_categorical_crossentropy
from   keras.optimizers import  adam
from   keras            import  callbacks as cb
DATASET    = "D:\\Dataset2.csv"
TARGETFILE = "D:\\Y2.csv"
WINDOW     = 2600
seed       = 4500
X          = np.loadtxt(DATASET, delimiter=",")
X          = X[:, 0:WINDOW]
scaler     = pp.MaxAbsScaler()
X_scaled   = scaler.fit_transform(X)
X_abs      = np.absolute(X)
y_array    = np.loadtxt(TARGETFILE, delimiter=",")
y          = np_utils.to_categorical(y_array)
LS         = 834
RS         = 4500
X2         = np.absolute(X_scaled)
Length     = X.shape[1]
#######################Fully connected model##################
My_model = Sequential()
My_model.add(Dense(int(Length/4),activation='relu',
input_shape=(Length,)))
My_model.add(Dense(int(Length/16), activation='relu'))
My_model.add(Dense(int(Length/32), activation='relu'))
My_model.add(Dropout(rate=0.4))
My_model.add(Dense(int(Length/64),  activation='relu'))
My_model.add(Dense(int(Length/128),  activation='relu'))
My_model.add(Dense(9, activation='softmax'))
My_model.summary()
My_model.compile(optimizer=adam(),
                 loss=categorical_crossentropy, metrics=['accuracy'] )
###############################################################
plot_model(My_model, to_file='D:\\FC_model.png' ,
           show_shapes=True, show_layer_names=True)
logger           = cb.CSVLogger("D:\\logger.log")
model_checkpoint = cb.ModelCheckpoint("D:\\Model.h5")
tensorboard      = cb.TensorBoard(log_dir="D:\\LOGS")
call_backs = [ logger,model_checkpoint, tensorboard ]
#################### training Model ########################
Network_history= My_model.fit(X2,y, epochs=200,
                 validation_split=0.15,batch_size=45,callbacks=call_backs)
history  = Network_history.history
losses   = history['loss']
accuracy = history['accuracy']
################Plot loss and accuracy ######################
plt.figure()
plt.xlabel('Epochs')
plt.ylabel('losses,␣accuracy,␣val_losses,␣val_accuracy')
plt.plot(losses, linewidth=1, marker='o', markersize=4, color = 'b')
plt.plot(history['val_loss'],  linewidth=1, markersize=4,
         marker='s', color = 'orange')
plt.plot(accuracy, linewidth=2, marker = '^', markersize=4,color = 'g')
plt.plot(history['val_accuracy'], linewidth=2, marker='d',
         markersize=4, color='r')
plt.legend(['loss','val_loss', 'accuracy', 'val_acc'])
My_model.save("D:\\FCModel.h5")
My_model.save_weights("D:\\FCModel_weights.h5")
```

A.7 Python code for 1D-CNN Model 1 with two convolutional layer

```
"""
Created on Fri Dec  6 11:43:20 2019
    @author: Ali Mehrabi
    Model 1, 1D CNN
"""
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from   sklearn.model_selection import train_test_split
from   sklearn.preprocessing   import StandardScaler
from   keras.utils import np_utils, plot_model
from   keras.models import Sequential, Model
from   keras.layers import Dense,Dropout,Conv1D,Flatten,
       AveragePooling1D,Input
from   keras.losses import categorical_crossentropy
from   keras.optimizers import SGD, adam
from   keras            import  callbacks as cb
from   sklearn          import preprocessing  as pp

DATASET    = "D:\\Dataset2.csv"
TARGETFILE = "D:\\Y2.csv"
WINDOW     = 2600
seed       = 9
X          = np.loadtxt(DATASET, delimiter=",")
X          = X[:, 0:WINDOW]
Length     = X.shape[1]
scaler     = pp.MaxAbsScaler()
X_scaled   = scaler.fit_transform(X)
X_abs      = np.absolute(X)
y_array    = np.loadtxt(TARGETFILE, delimiter=",")
y          = np_utils.to_categorical(y_array)
LS         = 834
RS         = 2500
X2         = np.absolute(X_scaled)
XX         = np.expand_dims(X2, axis=2)
#################### Creating Model#######################
X_train, X_test, y_train,y_test =
 train_test_split(XX,y, test_size=0.2, random_state=seed)
############################################################
######################### Parameters ######################
epochs     = 80
batch_size = 20 #X_train.shape[0]
channels   = 1
verbose    = 0
n_samples, n_features = X.shape[0], X.shape[1]
###########################################################
My_Input  = Input(shape=(n_features,1))
Conv1     = Conv1D(filters=325, kernel_size=16,
                   strides= 8, activation='relu')(My_Input)
Conv2     = Conv1D(filters=110, kernel_size=8,
                   strides= 1, activation='relu')(Conv1)
AVG1      = AveragePooling1D()(Conv2)
Flat1     = Flatten()(AVG1)
Out_layer = Dense(9,   activation='softmax')(Flat1)
My_model  = Model(My_Input, Out_layer)
My_model.summary()
My_model.compile(optimizer=adam(),
                 loss=categorical_crossentropy, metrics=['accuracy'] )
##########################################################
logger           = cb.CSVLogger("D:\\CONV1D_logger.log")
model_checkpoint = cb.ModelCheckpoint("D:\\CONV1D_model_checkpoint.h5")
```

```
tensorboard      = cb.TensorBoard(log_dir="D:\\LOGS")
call_backs       = [ logger,model_checkpoint, tensorboard ]
############################################################
## training Model
Network_history = My_model.fit(XX,y, epochs=epochs,
                  validation_data=(X_test,y_test), callbacks=call_backs)
history = Network_history.history
###########plot loss and accuracy ######################
losses   = history['loss']
accuracy = history['accuracy']
plt.figure()
plt.xlabel('Epochs')
plt.ylabel('losses,␣accuracy')
plt.plot(losses)
plt.plot(accuracy)
plt.legend(['loss','accuracy'])#
plt.figure()
plt.xlabel('Epochs')
plt.ylabel('val_losses,␣val_accuracy')
plt.plot(history['val_loss'])
plt.plot(history['val_accuracy'])
plt.legend(['val_loss', 'val_acc'])
###########################################################################
plt.figure()
plt.xlabel('Epochs')
plt.ylabel('losses,␣val_losses')
plt.plot(history['loss'])
plt.plot(history['val_loss'])
plt.legend(['loss', 'val_loss'])
###########################################################################
plt.figure()
plt.xlabel('Epochs')
plt.ylabel('accuracy,␣val_accuracy')
plt.plot(history['accuracy'])
plt.plot(history['val_accuracy'])
plt.legend(['accuracy', 'val_accuracy'])
### evaluate model
test_loss, test_accuracy = My_model.evaluate(XX,y)
print(My_model.metrics_names[1], test_accuracy*100)
print(My_model.metrics_names[0], test_loss)
plt.figure()
plt.xlabel('Epochs')
plt.ylabel('losses,␣accuracy,␣val_losses,␣val_accuracy')
#plt.plot(losses, marker='o', color = 'b')
#plt.plot(history['val_loss'], marker='s', color = 'orange')
plt.plot(accuracy, marker = '^', color = 'g')
plt.plot(history['val_accuracy'], marker='d', color='r')
plt.legend(['loss','val_loss', 'accuracy', 'val_acc'])#
############################### PLOT MODEL ###############################
plot_model(My_model, to_file='D:\\CONV1D_model.png' ,
           show_shapes=True, show_layer_names=True)
```

```
############################ SAVE MODEL ##################################
My_model.save("D:\\CONV1D_Model.h5")
My_model.save_weights("D:\\CONV1D_Model_weights.h5")
```

A.8 Python code for 1D-CNN Model 2 with one convolutional layer

```
"""
Created on Fri Dec  6 11:43:20 2019
@author: Ali Mehrabi
Model 2,  1D CNN
"""
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from   sklearn.model_selection import train_test_split
from   sklearn.preprocessing   import StandardScaler
from   keras.utils import np_utils, plot_model
from   keras.models import Sequential, Model
from   keras.layers import Dense, Dropout,Conv1D,MaxPool1D,Flatten,Input
from   keras.losses import categorical_crossentropy
from   keras.optimizers import SGD, adam
from   keras            import  callbacks as cb
from   sklearn          import preprocessing  as pp

DATASET    = "D:\\Dataset2.csv"
TARGETFILE = "D:\\Y2.csv"
WINDOW     = 2600
seed       = 9
X          = np.loadtxt(DATASET, delimiter=",")
X          = X[:, 0:WINDOW]
Length     = X.shape[1]
scaler     = pp.MaxAbsScaler()
X_scaled   = scaler.fit_transform(X)
X_abs      = np.absolute(X)
y_array    = np.loadtxt(TARGETFILE, delimiter=",")
y          = np_utils.to_categorical(y_array)
LS         = 834
RS         = 2500
X2         = np.absolute(X_scaled[:,1000:1400])
XX         = np.expand_dims(X2, axis=2)
############################## Creating Model #############
X_train, X_test, y_train,y_test =
train_test_split(XX,y, test_size=0.2, random_state=seed)
#########################################################
######################## Parameters #####################
epochs     = 250
batch_size = 20 #X_train.shape[0]
channels   = 1
verbose    = 0
```

```
n_samples, n_features = X2.shape[0], X2.shape[1]
###########################################################
My_Input  = Input(shape=(n_features,1))
Conv1     = Conv1D(filters=250, kernel_size=16,
            strides= 8, activation='relu')(My_Input)
AVG1      = MaxPool1D()(Conv1)
Flat1     = Flatten()(AVG1)
Out_layer = Dense(9,   activation='softmax')(Flat1)
My_model  = Model(My_Input, Out_layer)
My_model.summary()
My_model.compile(optimizer=adam(),
           loss=categorical_crossentropy, metrics=['accuracy'])
################################################################
logger           = cb.CSVLogger("D:\\CONV1D_logger.csv")
model_checkpoint = cb.ModelCheckpoint("D:\\CONV1D_model_checkpoint.h5")
tensorboard      = cb.TensorBoard(log_dir="D:\\LOGS")
call_backs       = [ logger,model_checkpoint, tensorboard ]
################################################################
## training Model
Network_history = My_model.fit(XX,y, epochs=epochs,
                  validation_data=(X_test,y_test), callbacks=call_backs)
history = Network_history.history
##############################plot loss and accuracy######################
losses   = history['loss']
accuracy = history['accuracy']
plt.figure()
plt.xlabel('Epochs')
plt.ylabel('losses,␣accuracy')
plt.plot(losses)
plt.plot(accuracy)
plt.legend(['loss','accuracy'])#
plt.figure()
plt.xlabel('Epochs')
plt.ylabel('val_losses,␣val_accuracy')
plt.plot(history['val_loss'])
plt.plot(history['val_accuracy'])
plt.legend(['val_loss', 'val_acc'])
################################### PLOT MODEL ###########################
plot_model(My_model, to_file='D:\\CONV1D_model.png',
           show_shapes=True, show_layer_names=True)
################################### SAVE MODEL ###########################
My_model.save("D:\\CONV1D_Model.h5")
My_model.save_weights("D:\\CONV1D_Model_weights.h5")
```

B
Appendix B

B.1 VHDL package for RNS Montgomery reduction

```
-----------------------------------------------------------------
--
-- Written by: Ali Mehrabi
-- Create Date: 13.02.2019 11:34:32
-- Design Name:
-- Module Name: MNG_PACK - Behavioral
-- Project Name: RNS ECC
-- Target Devices: VIRTX 7
-- Tool Versions:
-- Description: The VHDL package defines types,
--              constants and functions used in
--              implementation of RNS Montgomery reduction unit.
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

package MNG_PACKAGE is

constant w              : INTEGER := 66;
constant channel_width  : INTEGER := w;
constant total_channels : INTEGER := 8;
constant c              : INTEGER := 4;

constant KEY            : std_logic_vector(255 downto 0) :=
X"8000_0000_0000_0000_0000_0000_0000_0000
```

```
_0000_0000_0000_0000_0000_0000_0000_0009";
--------------------------------------- TYPE declaration
type     type_RNS       is array (1 to total_channels/2)
of std_logic_vector(w-1 downto 0);  -- 66 bits
type     type_DRNS      is array (1 to total_channels/2)
of std_logic_vector(2*w-1 downto 0); -- 2* 66 bits
type     type_CRNS      is array (1 to total_channels )
of std_logic_vector(w-1 downto 0);  -- 66 bits
type     type_RNS_Table is array (1 to total_channels/2)
of type_RNS;
type     type_RSUM      is array (1 to total_channels/2)
of std_logic_vector(w downto 0);
type     type_YACU      is array (1 to total_channels/2)
of std_logic_vector(w+2 downto 0);
---------------------------------------------------------------------
--moduli set:
--2^66-1     ;  2^66-2^2-1 ;
--2^66-2^3-1 ;  2^66-2^4-1 ;
--2^66-2^5-1 ;  2^66-2^6-1 ;
--2^66-2^8-1;   2^66-2^9-1 ;
--
constant Moduli : type_CRNS := (
"11"&X"FFFFFFFFFFFFFFFF","11"&X"FFFFFFFFFFFFFFFB",
"11"&X"FFFFFFFFFFFFFFF7","11"&X"FFFFFFFFFFFFFFEF",
"11"&X"FFFFFFFFFFFFFFDF","11"&X"FFFFFFFFFFFFFFBF",
"11"&X"FFFFFFFFFFFFFEFF","11"&X"FFFFFFFFFFFFFDFF"
);
constant CModuli : type_CRNS := (
"00"&X"0000000000000001", "00"&X"0000000000000005",
"00"&X"0000000000000009", "00"&X"0000000000000011",
"00"&X"0000000000000021", "00"&X"0000000000000041",
"00"&X"0000000000000101", "00"&X"0000000000000201");

constant K : type_RNS := (
"11"&X"FFFFFFFFFFFFFFFF","11"&X"FFFFFFFFFFFFFFFB",
"11"&X"FFFFFFFFFFFFFFF7","11"&X"FFFFFFFFFFFFFFEF"
);

constant Q : type_RNS := (
"11"&X"FFFFFFFFFFFFFFDF","11"&X"FFFFFFFFFFFFFFBF",
"11"&X"FFFFFFFFFFFFFEFF","11"&X"FFFFFFFFFFFFFDFF"
);
-----------------------------------------------------------------------------
constant KINV : type_RNS := (
"11"&X"FDFFFFFFFFFFFFFF", "00"&X"4555555555555555", "11"&X"1BFFFFFFFFFFFFF9",
"00"&X"A0AAAAAAAAAAAAA8");
constant QINV : type_RNS := (
"10"&X"0494A5CA5CA5CA4C", "00"&X"0180186186186186", "00"&X"B0C91E79E79E797B",
"00"&X"4922235A35A35A11");
------------------------------------------------------------------------------
constant IPNQ : type_RNS := (
"10"&X"52B3070CAE4A8251", "11"&X"528B437349C54583", "01"&X"93AB34988ACE6E6B",
"10"&X"6A0DAB5A85884853");
constant PNQ : type_RNS := (
"01"&X"0DE641B8427554D2", "01"&X"10EA1A5D70599036", "01"&X"7BE4E887474BAEE9",
"10"&X"4106273148516B36");
constant PK : type_RNS := (
"00"&X"03FFFFEFFFFFC2F", "01"&X"F3FFFFEFFFFFC2F", "11"&X"63FFFFEFFFFFC41",
"00"&X"C3FFFFFEFFFFFD72");
constant PNK : type_RNS := (
"00"&X"1BB20C7BD99A0D63", "01"&X"AB4C56608CDFB866", "11"&X"E802AF45DAEDE46C",
"01"&X"C0657C4B03C6B1C5");
constant HNK : type_RNS := (
"00"&X"0000004000000000", "01"&X"13D5045218043984", "01"&X"5773EB99F5007171",

"00"&X"3C2620AA4CB54F75");
constant IHNK : type_RNS := (
"11"&X"FFFFFFFDFFFFFFF", "10"&X"6C1A715B1820059F", "11"&X"5AA88C14660AFF87",
"11"&X"6E0A065AC70CC8D4");
constant KIJ : type_Rns_Table:=(
("00"&X"0000000000002A00", "00"&X"0000000000027600", "00"&X"0000000000E4DE00",
"00"&X"000000000791BE00"),
("00"&X"0000000000003000", "00"&X"000000000002A000", "00"&X"0000000000E88000",
"00"&X"0000000007A10000"),
("00"&X"0000000000003800", "00"&X"000000000002D000", "00"&X"0000000000EC4000",
"00"&X"0000000007B08000"),
("00"&X"0000000000005400", "00"&X"0000000000034800", "00"&X"0000000000F42000",
"00"&X"0000000007D04000")
);
constant QIJ : type_Rns_Table:=(
("11"&X"FFFFFFFFF7FFFFF", "11"&X"FFFFFFFFFF8ACC3B", "11"&X"FFFFFFFFFF9531F7",

"11"&X"FFFFFFFFFFA8CFEF"),
("11"&X"FFFFFFFFFFBFFFFF", "11"&X"FFFFFFFFFFC94E3B", "11"&X"FFFFFFFFFFD239F7",

"11"&X"FFFFFFFFFFE2EFEF"),
("11"&X"FFFFFFFFFFEFFFFF", "11"&X"FFFFFFFFFFF2FA3B", "11"&X"FFFFFFFFFFF5A9F7",

"11"&X"FFFFFFFFFFFA2FEF"),
("11"&X"FFFFFFFFFFF7FFFF", "11"&X"FFFFFFFFFFF98A3B", "11"&X"FFFFFFFFFFFAE9F7",

 "11"&X"FFFFFFFFFFFD2FEF")
);
constant THETA: type_Rns_Table:=(
("10"&X"1FE000000800001E","01"&X"C99249249B6DB6FC","01"&X"068000000AAAAAD0",
"11"&X"73C000001000001A"),
("01"&X"0FF000000400000F","00"&X"D58888888CCCCCDC","10"&X"B9A4924929249254",
"10"&X"7BEAAAAAB0000003"),
("11"&X"43FC000001000003","00"&X"32D75D75D861861C","01"&X"C6D294A52A5294A5",
"01"&X"4BFBBBBBBCCCCCCA"),
("11"&X"A1FE000000800001","00"&X"09183060C2040812","01"&X"3112492492CB2CB2",
"11"&X"3548421084A5293E")
);
constant PQNIJ: type_Rns_Table:=(
("10"&X"1FE000000800001E", "01"&X"0FF000000400000F", "11"&X"43FC000001000003",
"11"&X"A1FE000000800001"),
("01"&X"C99249249B6DB6FC", "00"&X"D58888888CCCCCDC", "00"&X"32D75D75D861861C",
"00"&X"09183060C2040812"),
("01"&X"068000000AAAAAD0", "10"&X"B9A4924929249254", "01"&X"C6D294A52A5294A5",
"01"&X"3112492492CB2CB2"),
("11"&X"73C000001000001A", "10"&X"7BEAAAAAB0000003", "01"&X"4BFBBBBBBCCCCCCA",
"11"&X"3548421084A5293E")
);

constant alphaP : type_Rns_Table:=(
("00"&X"0000000000000000", "00"&X"0000000000000000", "00"&X"0000000000000000",
"00"&X"0000000000000000"),
("11"&X"FC000001000003D0", "10"&X"0C000001000003CC", "00"&X"9C000001000003B6",
"11"&X"3C0000010000027D"),
("11"&X"F8000002000007A1", "00"&X"180000020000079D", "01"&X"380000020000076C",
"10"&X"780000020000050B"),
("11"&X"F400000300000B72", "10"&X"2400000300000B69",  "01"&X"D400000300000B22",
"01"&X"B400000300000799")
);
constant alphaQ : type_Rns_Table:=(
("00"&X"0000000000000000", "00"&X"0000000000000000", "00"&X"0000000000000000",
"00"&X"0000000000000000"),
("11"&X"FFFFFFFEFFFFFFFF", "11"&X"FFFFFFFFF32E56FB", "11"&X"FFFFFFFFF5FCAFF7",
"11"&X"FFFFFFFFFA8CFFEF"),
("11"&X"FFFFFFFDFFFFFFFF", "11"&X"FFFFFFFFE65CADFB", "11"&X"FFFFFFFFEBF95FF7",
"11"&X"FFFFFFFFF519FFEF"),
("11"&X"FFFFFFFCFFFFFFFF", "11"&X"FFFFFFFFD98B04FB", "11"&X"FFFFFFFFE1F60FF7",
"11"&X"FFFFFFFFEFA6FFEF")
);
constant alphaK : type_Rns_Table:=(
("00"&X"0000000000000000", "00"&X"0000000000000000", "00"&X"0000000000000000",
"00"&X"0000000000000000"),
("11"&X"FFFFFFFFFFFABFDF", "11"&X"FFFFFFFFFF627FBF", "11"&X"FFFFFFFF1B21FEFF",
"11"&X"FFFFFFF0DC83FDFF"),
("11"&X"FFFFFFFFFFF57FDF", "11"&X"FFFFFFFFFEC4FFBF", "11"&X"FFFFFFFE3643FEFF",
"11"&X"FFFFFFE1B907FDFF"),
("11"&X"FFFFFFFFFFF03FDF", "11"&X"FFFFFFFFFE277FBF", "11"&X"FFFFFFFD5165FEFF",
"11"&X"FFFFFFD2958BFDFF")
);
constant QNK : type_RNS := (
"00"&X"0000040000000000", "01"&X"13D5045218043984",
"01"&X"5773EB99F5007171", "00"&X"3C2620AA4CB54F75"
);
-----------------------------------------------------------------------------
procedure MMR_M1(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M2(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M3(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M4(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M5(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M6(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M7(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M8(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
-------------------------------------------------------------------------
```

```
function BLOCK_SUM(ARNS,BRNS: in type_RNS ; F : std_logic)
return type_RNS;
function BLOCK_SUMK(ARNS,BRNS: in type_RNS)
return type_RNS;
function BLOCK_SUMQ(ARNS,BRNS: in type_RNS)
return type_RNS;
function Read_AlphaK(i: std_logic_vector)
return type_RNS;
function Read_AlphaP(i: std_logic_vector)
return type_RNS;
function Read_AlphaQ(i: std_logic_vector)
return type_RNS;
function  REDSUM( Ai: type_YACU; F: std_logic)
return type_RNS;
end package;
package body MNG_PACKAGE is
--------------------------------------------------------------------------
procedure MMR_M1(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH : std_logic_vector(w-1 downto 0);
variable N     : std_logic_vector(w downto 0);
variable M     : std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(47 downto 0):=(others=>'0');
begin
M:= Moduli(1);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
N:= '0'&AH+AL;
 if (N>= M) then
    N(w downto 0):= N(w downto 0)-M(w-1 downto 0);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M1;
--------------------------------------------------------------------------
procedure MMR_M2(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM  : std_logic_vector(w downto 0);
variable N,Z : std_logic_vector(w+1 downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(61 downto 0):=(others=>'0');
begin
M:= Moduli(2);
AH(w-1 downto 0) := A(2*w-1 downto w);
AL(w-1 downto 0) := A(w-1 downto 0);
AAL(w-1 downto 0):= A(2*w-3 downto w)&"00";
AAH(w-1 downto 0):= Z0&(A(2*w-1 downto 2*w-2)&A(2*w-1 downto 2*w-2));
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM - M;
end if;
N:= '0'&AM+AAL+AAH;
 if (N>=Moduli(2)) then
    N(w+1 downto 0):= N(w+1 downto 0)-moduli(2);
 else
    N(w+1 downto 0):= N(w+1 downto 0);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M2;
--------------------------------------------------------------------------
procedure MMR_M3(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM : std_logic_vector(w downto 0);
variable N  : std_logic_vector(w+1 downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(59 downto 0):=(others=>'0');
begin
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
M:= Moduli(3);
if AM >= M then
   AM:= AM - M;
end if;
AAL:= AH(w-4 downto 0)&"000";
AAH:= Z0&AH(w-1 downto w-3)&AH(w-1 downto w-3);
N:= '0'&AM+ AAH+AAL;
 if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-Moduli(3);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M3;
------------------------------------------------------------------------
procedure MMR_M4(
signal A     :  in  std_logic_vector(2*w-1 downto 0);
signal R     :  out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM   :  std_logic_vector(w   downto 0);
variable N    :  std_logic_vector(w+1 downto 0);
variable M    :  std_logic_vector(w-1 downto 0);
constant Z0   :  std_logic_vector(w-9 downto 0):=(others=>'0');
begin
M:= Moduli(4);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM - M;
end if;
AAL:= A(2*w-5 downto w)&"0000";
AAH:= Z0&A(2*w-1 downto 2*w-4)&A(2*w-1 downto 2*w-4);
N(w+1 downto 0) := '0'&AM+AAL+AAH;
 if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M4;
------------------------------------------------------------
procedure MMR_M5(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM: std_logic_vector(w downto 0);
variable N : std_logic_vector(w+1 downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(w-11 downto 0):=(others=>'0');
begin
M:= Moduli(5);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM - M;
end if;
AAL:= A(2*w-6 downto w)&"00000";
AAH:= Z0&A(2*w-1 downto 2*w-5)&A(2*w-1 downto 2*w-5);
N:= '0'&AM+AAL+AAH;
 if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
 else
    N(w+1 downto 0):= N(w+1 downto 0);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M5;
-----------------------------------------------------------------------
procedure MMR_M6(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM  : std_logic_vector(w downto 0);
variable N   : std_logic_vector(w+1 downto 0);
variable M   : std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(w-13 downto 0):=(others=>'0');
begin
M:= Moduli(6);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM-M;
end if;
AAL:= A(2*w-7 downto w)&"000000";
AAH:= Z0&A(2*w-1 downto 2*w-6)&A(2*w-1 downto 2*w-6);
N:= '0'&AM+AAL+AAH;
if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M6;
------------------------------------------------------------------------
procedure MMR_M7(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM  : std_logic_vector(w downto 0);
variable N   : std_logic_vector(w+1 downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(w-17 downto 0):=(others=>'0');
begin
M := Moduli(7);
```

```
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM := AM-M;
end if;
AAL:= A(2*w-9 downto w)&"00000000";
AAH:= Z0&A(2*w-1 downto 2*w-8)&A(2*w-1 downto 2*w-8);
N:= '0'&AM+AAL+AAH;
if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M7;
-------------------------------------------------------------------
procedure MMR_M8(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM   : std_logic_vector(w downto 0);
variable N    : std_logic_vector(w+1 downto 0);
variable M    : std_logic_vector(w-1  downto 0);
constant Z0   : std_logic_vector(w-19 downto 0):=(others=>'0');
begin
M:= Moduli(8);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM := AM-M;
end if;
AAL:= A(2*w-10 downto w)&"000000000";
AAH:= Z0&A(2*w-1 downto 2*w-9)&A(2*w-1 downto 2*w-9);
N:= '0'&AM+AAL+AAH;
if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M8;

function BLOCK_SUM(ARNS,BRNS: in type_RNS ; F : std_logic)
return type_RNS is
variable j: integer;
variable C: type_RNS;
variable A,B: std_logic_vector(w-1 downto 0):=(others=>'0');
variable Sum,S: std_logic_vector(w downto 0):=(others=>'0');
begin
for j in 1 to total_channels/2 loop
A:=(ARNS(j));
B:=(BRNS(j));
Sum:= '0'&A+B;
case F is
when '0' =>
if Sum >= Moduli(j) then
   S:= Sum - Moduli(j);
else
   S:= Sum;
end if;
```

```
when others  =>
if Sum >= Moduli(j+4) then
   S:= Sum - Moduli(j+4);
else
   S:= Sum;
end if;
end case;
C(j):= S(w-1 downto 0);
end loop;
return C;
end function;

function BLOCK_SUMK(ARNS,BRNS: in type_RNS)
return type_RNS is
variable j: integer;
variable C: type_RNS;
variable Sum,S,A,B: std_logic_vector(w downto 0):=(others=>'0');
begin
for j in 1 to total_channels/2 loop
A:=('0'& ARNS(j));
B:=('0'& BRNS(j));
Sum:= A+B;
if Sum >= K(j) then
   S:= Sum - K(j);
else
   S:= Sum;
end if;
C(j):= S(w-1 downto 0);
end loop;
return C;
end function;

function BLOCK_SUMQ(ARNS,BRNS: in type_RNS)
return type_RNS is
variable j: integer;
variable C: type_RNS;
variable Sum,S,A,B: std_logic_vector(w downto 0):=(others=>'0');
begin
for j in 1 to total_channels/2 loop
A:=('0'& ARNS(j));
B:=('0'& BRNS(j));
Sum:= A+B;
if Sum >= Q(j) then
   S:= Sum - Q(j);
else
   S:= Sum;
end if;
C(j):= S(w-1 downto 0);
end loop;
return C;
end function;

function Read_AlphaK(i: std_logic_vector)
return type_RNS is
variable j: integer range 0 to total_channels/2-1;
begin
if I >=0 and I<= (total_channels/2-1) then
   j:=conv_integer(I+1);
```

```
else
   j:=1;
end if;
return AlphaK(j);
end function Read_AlphaK;

function Read_AlphaQ(i: std_logic_vector)
return type_RNS is
variable j: integer range 0 to total_channels/2-1;
begin
if I >=0 and I<= (total_channels/2-1) then
   j:=conv_integer(I+1);
else
   j:=1;
end if;
return AlphaQ(j);
end function Read_AlphaQ;

function Read_AlphaP(i: std_logic_vector)
return type_RNS is
variable j: integer range 0 to total_channels/2-1;
begin
if I >=0 and I<= (total_channels/2-1) then
   j:=conv_integer(I+1);
else
   j:=1;
end if;
return AlphaP(j);
end function Read_AlphaP;

function REDSUM( Ai: type_YACU; F: std_logic)
return type_RNS is
variable E : std_logic_vector(w+2 downto 0);
variable R : type_RNS;
variable KK,QQ : type_RSUM;
begin
for i in 1 to 4 loop
KK(i) := K(i)&'0';
QQ(i) := Q(i)&'0';
if F='0' then
     if    (Ai(i) >= KK(i)) then
            E := Ai(i) - KK(i);
     elsif (Ai(i) >= K(i))  then
            E := Ai(i) - K(i);
     else
            E:= Ai(i);
     end if;
elsif F='1' then
    if    (Ai(i) >= QQ(i)) then
           E := Ai(i) - QQ(i);
    elsif (Ai(i) >= Q(i))     then
           E := Ai(i) - Q(i);
    else
           E:= Ai(i);
    end if;
end if;
R(i)(w-1 downto 0) := E(w-1 downto 0);
end loop;
```

```
return R;
end function REDSUM;
end package body MNG_PACKAGE;
```

B.2 Hardware description for RNS Motgomery reduction

```
--------------------------------------------------------------------
-- Written by: Ali Mehrabi
-- Create Date: 14.02.2019 11:01:13
-- Design Name:
-- Module Name: MNG_RED - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------
library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all; -- for multiply operation
use IEEE.std_logic_arith.all;    -- for std_logic to integer conversion
use WORK.MNG_PACKAGE.all;


entity MNG_RED is
Port ( CLK : IN  std_logic;
       RST : IN  std_logic;
       A   : IN  type_CRNS;
       B   : IN  type_CRNS;
       D   : OUT type_CRNS
      );
end MNG_RED;


architecture Behavioral of MNG_RED is
type STATE is (S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,
S15,S16,S17,S18,S19,S20);
signal ST : STATE;
signal AK : type_RNS ;
signal AQ : type_RNS ;
signal BK : type_RNS ;
signal BQ : type_RNS ;
signal A1,B1,D1  : type_RNS;
signal A2,B2,D2  : type_RNS;
signal G ,L1,L2  : type_RNS;
signal A3,B3,S   : type_RNS;
signal YI1,YI2   : type_RNS;
signal C1,C2,F : std_logic;
signal Alpha : std_logic_vector(1 downto 0);
```

```
component RNSMULT is
Port ( A: IN  type_RNS;
       B: IN  type_RNS;
       C: IN  std_logic;
       D: OUT type_RNS);
end component RNSMULT;

component RNSSUM is
Port ( A: IN  type_RNS;
       B: IN  type_RNS;
       F: IN  std_logic;
       S: OUT type_RNS);
end component RNSSUM;

component G_adder is
Port (A1,A2,A3,A4 : in std_logic_vector(3 downto 0);
      S : out std_logic_vector(1 downto 0)
      );
end component G_adder;

begin
AK <= (A(1),A(2),A(3),A(4));
AQ <= (A(5),A(6),A(7),A(8));
BK <= (B(1),B(2),B(3),B(4));
BQ <= (B(5),B(6),B(7),B(8));

U1: RNSMULT port map ( A=> A1, B=>B1, C=>C1, D=> D1);
U2: RNSMULT port map ( A=> A2, B=>B2, C=>C2, D=> D2);
U3: RNSSUM  port map ( A=> A3, B=>B3, F=>F , S=> S );
U4: G_ADDER port map ( A1 =>G(1)(w-1 downto w-4), A2=> G(2)(w-1 downto w-4),
                       A3 =>G(3)(w-1 downto w-4), A4=> G(4)(w-1 downto w-4),S=> Alpha);
process(CLK, RST)
begin
if RST = '0' then
   ST <= S0;
   C1 <= '0'; -- multiplier 1 reduction in K
   C2 <= '1'; -- multiplier 2 reduction in Q
   F  <= '1';
   D <= (others=>(others=>'0'));
elsif CLK='1' and CLK'event then
case ST is
     when S0 =>
          A1 <= AK;
          B1 <= BK;
          A2 <= AQ;
          B2 <= BQ;
          ST <= S1;
     when S1 =>
          ST <= S2;
     when S2 =>
          A1 <= D1;    --  AK.BK in D1
          B1 <= QNK;   --  Q^-1
          A2 <= D2;    --  AQ.BQ in D2
          B2 <= IPNQ;  --  AQ.BQ*IPNQ
          ST <= S3;
     when S3 =>
          C1 <= '0';  -- reduction in K from here
          C2 <= '0';
          ST  <= S4;
     when S4 =>
          G   <= D2; -- Gammaq = AQ.BQ.IPNQ in register G
          L1  <= D1; -- AK.BK.PNK in register  L1
          A1  <= (D2(1),D2(1),D2(1),D2(1));
          B1  <= THETA(1);    --------------- Gamma1 * QIJ(1)(j)
          A2  <= (D2(2),D2(2),D2(2),D2(2));
          B2  <= THETA(2);    --------------- Gamma2 * QIJ(2)(j)
          ST  <= S5;
     when S5 =>
          F <= '0';
          ST  <= S6;
     when S6 =>
          A3  <= D1; --
          B3  <= D2; -- ADD D1 +D2
          L2  <= Read_AlphaP(alpha);  -- -alphaQ in L2
          A1  <= (G(3),G(3),G(3),G(3)); --------------- Gamma3 * QIJ(3)(j)
          B1  <= THETA(3);
          A2  <= (G(4),G(4),G(4),G(4)); -------------- Gamma4 * QIJ(4)(j)
          B2  <= THETA(4);
          ST  <= S7;
     when S7 =>
          L1  <= BLOCK_SUMK(L1,L2);
          YI1 <= S;   -- D1+D2
          ST  <= S8;
     when S8 =>
          C1<= '0';
          C2<= '0';
          A3  <= D1;
          B3  <= D2;
          ST  <= S9;
     when S9 =>
          A3  <= S;
          B3  <= YI1;
          ST  <= S10;
     when S10 =>
          A3  <= S;
          B3  <= L1;
          ST  <= S11;
     when S11 =>
          D(1)  <= S(1);
          D(2)  <= S(2);
          D(3)  <= S(3);
          D(4)  <= S(4);
          A2  <= S;
          B2  <= KINV;
          ST  <= S12;
     when S12 =>
          C1  <= '1';
          C2  <= '1';
          F   <= '1';
          ST  <= S13;
     when S13 =>
          G  <= D2;
          A1  <= (D2(1),D2(1),D2(1),D2(1));
          B1  <= KIJ(1);
          A2  <= (D2(2),D2(2),D2(2),D2(2));
          B2  <= KIJ(2);
```

```
            ST <= S14;
        when S14 =>
             ST <= S15 ;
        when S15 =>
             A3 <= D1;
             B3 <= D2;
             L2 <= Read_AlphaK(alpha);
             A1 <= (G(3),G(3),G(3),G(3));
             B1 <= KIJ(3);
             A2 <= (G(4),G(4),G(4),G(4));
             B2 <= KIJ(4);
             ST <= S16;
        when S16 =>
             YI1 <= S;
             ST <= S17;
        when S17 =>
             A3 <= D1;
             B3 <= D2;
             ST <= S18;
        when S18 =>
             A3 <= S;
             B3 <= L2;
             ST <= S19;
        when S19 =>
             A3 <= S;
             B3 <= YI1;
             ST <= S20;
        when S20 =>
             D(5) <= S(1);
             D(6) <= S(2);
             D(7) <= S(3);
             D(8) <= S(4);
             ST <= S20;
end case;
end if;
end process;
end Behavioral;

----------------------------------------------------------------------------------
-- Written by: Ali Mehrabi
-- Create Date: 13.02.2019 13:57:22
-- Design Name: RNS Multiplier
-- Module Name: RNS_MULT - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
----------------------------------------------------------------------------------


library IEEE;
library WORK;
```

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all; -- for multiply operation
use IEEE.std_logic_arith.all;    -- for std_logic to integer conversion
use WORK.MNG_PACKAGE.all;


entity RNSMULT is
Port ( A: IN  type_RNS;
       B: IN  type_RNS;
       C: IN  std_logic;
       D: OUT type_RNS);
end RNSMULT;

architecture Behavioral of RNSMULT is
signal M: type_DRNS;
signal R,S: type_RNS;
begin

RNSMUL: for i in 1 to total_channels/2 generate
        M(i) <= A(i)*B(i);
        end generate;
process(M)
begin
case C is
when '0' =>
MMR_M1(M(1),D(1));
MMR_M2(M(2),D(2));
MMR_M3(M(3),D(3));
MMR_M4(M(4),D(4));
when others =>
MMR_M5(M(1),D(1));
MMR_M6(M(2),D(2));
MMR_M7(M(3),D(3));
MMR_M8(M(4),D(4));
end case;
end process;
end Behavioral;
---------------------------------------------------------------
-- Written by: Ali Mehrabi
-- Create Date: 15.02.2019 10:06:58
-- Design Name:
-- Module Name: RNS_SUM - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------
library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all; -- for multiply operation
use IEEE.std_logic_arith.all;    -- for std_logic to integer conversion
use WORK.MNG_PACKAGE.all;


entity RNSSUM is
Port ( A: IN  type_RNS;
       B: IN  type_RNS;
       F: IN  std_logic;
       S: OUT type_RNS);
end RNSSUM;

architecture rtl of RNSSUM is
signal Si ,Ci : type_RSUM;
signal X,Y    : type_RSUM;

signal m : type_RNS;
begin
m(1) <= not moduli(1) when F='0' else not moduli(5);
m(2) <= not moduli(2) when F='0' else not moduli(6);
m(3) <= not moduli(3) when F='0' else not moduli(7);
m(4) <= not moduli(4) when F='0' else not moduli(8);
U:for i in 1 to total_channels/2 generate
Si(i) <= '0'&(A(i) xor B(i) xor m(i));
Ci(i) <= ((A(i) and B(i)) or (A(i) and m(i)) or (B(i) and m(i)))&'1';
X(i) <= Si(i)+ Ci(i);
Y(i) <= '0'&A(i) + B(i);
S(i) <= X(i)(w-1 downto 0) when Y(i)(w)='1' else Y(i)(w-1 downto 0);
end generate;
end architecture;
----------------------------------------------------------------------------------
-- Written by: Ali Mehrabi
-- Create Date: 14.02.2019 14:10:33
-- Design Name:
-- Module Name: G_ADDER - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.MNG_package.all;

entity G_adder is
Port (A1,A2,A3,A4 : in std_logic_vector(3 downto 0);
      S : out std_logic_vector(1 downto 0));
end G_adder;
```

```
architecture Behavioral of G_adder is

signal P10,P11,G10,G11 :std_logic_vector(3 downto 0);
signal S1    :std_logic_vector(5 downto 0);
signal S2    :std_logic_vector(6 downto 0);
signal C10   :std_logic_vector(5 downto 0);
signal C11   :std_logic_vector(5 downto 0);

begin
P10 <=  (A1 xor A2);
P11 <=  (A3 xor A4);
G10 <=  (A1 and A2);
G11 <=  (A3 and A4);
S1  <= "00"&(P10 xor P11); -- Delta = 2^4 added to S1
C10 <= '0'&((P10 and P11) or ((not G10) and G11) or (G10 and (not G11)))&'0';
C11 <= (G10 and G11) &"11";
S2 <= '0'&S1 + C10 + C11;
S(1 downto 0) <= S2(5 downto 4);
end Behavioral;
```

B.3 RNS Package for SOR Mudular reduction algorithm

```
--------------------------------------------------------------
-- Written by: Ali Mehrabi
-- Create Date: 22.06.2018 10:37:29
-- Design Name:
-- Module Name: RNS_PACKAGE - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
---------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

package RNS_PACKAGE is

constant w              : INTEGER := 66;
constant channel_width  : INTEGER := w;
constant total_channels : INTEGER := 8;
constant c              : INTEGER := 4;

constant KEY            : std_logic_vector(255 downto 0) :=
X"8000_0000_0000_0000_0000_0000_0000_0000
_0000_0000_0000_0000_0000_0000_0000_0001";
----------------------- TYPE declaration -----------------------------
type     type_RNS       is array (1 to total_channels) of
```

```
std_logic_vector(w-1 downto 0);
type     type_UDIM      is array (1 to total_channels) of
std_logic_vector(w-15 downto 0);
type     type_RNS_Table is array (1 to total_channels) of type_RNS;
type     type_YSUM      is array (1 to total_channels) of
std_logic_vector(w downto 0);
type     type_YACU      is array (1 to total_channels) of
std_logic_vector(w+2 downto 0);
type     type_DRNS      is array (1 to total_channels) of
std_logic_vector(2*w-1 downto 0);
---------------------------------------------------------------------
--moduli set:
--2^66-1     ;  2^66-2^2-1 ;
--2^66-2^3-1 ;  2^66-2^4-1 ;
--2^66-2^5-1 ;  2^66-2^6-1 ;
--2^66-2^8-1;   2^66-2^9-1 ;
--
constant Moduli : type_RNS := (
"11"&X"FFFFFFFFFFFFFFFF","11"&X"FFFFFFFFFFFFFFFB",
"11"&X"FFFFFFFFFFFFFFF7","11"&X"FFFFFFFFFFFFFFEF",
"11"&X"FFFFFFFFFFFFFFDF","11"&X"FFFFFFFFFFFFFFBF",
"11"&X"FFFFFFFFFFFFFEFF","11"&X"FFFFFFFFFFFFFDFF"
);

constant DModuli : type_YSUM := (
"111"&X"FFFFFFFFFFFFFFFE","111"&X"FFFFFFFFFFFFFFF6",
"111"&X"FFFFFFFFFFFFFFEE","111"&X"FFFFFFFFFFFFFFDE",
"111"&X"FFFFFFFFFFFFFFBE","111"&X"FFFFFFFFFFFFFF7E",
"111"&X"FFFFFFFFFFFFFDFE","111"&X"FFFFFFFFFFFFFBFE"
);
----------------------------------------------------------
constant CModuli : type_RNS := (
"00"&X"0000000000000001","00"&X"0000000000000005",
"00"&X"0000000000000009","00"&X"0000000000000011",
"00"&X"0000000000000021","00"&X"0000000000000041",
"00"&X"0000000000000101","00"&X"0000000000000201"
);
----------------------------------------------------------
constant DINV : type_RNS := (
"11"&X"FFFFFFFDFFFFFFF","10"&X"6C1A715B1820059F",
"11"&X"5AA88C14660AFF87","11"&X"6E0A065AC70CC8D4",
"00"&X"B7773DD32BC5056F","01"&X"13815F1861ADB944",
"00"&X"73AA4CA0D49099CD","00"&X"8C9012A978C4D8F2"
);

constant DiMmj : type_Rns_Table:=(
("10"&X"BE698F14B22F6A6B","10"&X"CC87376D7EC5D27F","10"&X"DA365FBC3307AB13",
"10"&X"F44F315789611FBB","11"&X"2394E04B6D33350B","11"&X"6FC0AB540F94FFAB",
"11"&X"D3F081BEF90DBF6B","11"&X"EA8E98DC1B2C146B"),
("10"&X"CC87376A914F74C3","10"&X"DA6D5FAC057452D7","10"&X"E7E587EF0DC942EB",
"11"&X"0191D9793E846B13","11"&X"30048799D8C75963","11"&X"7AA247E9B4BB9E03",
"11"&X"D819EB5728A119C3","11"&X"ECA81E79CECABEC3"),
("10"&X"DA365F9ECAB96D9B","10"&X"E7E587D5AEE3E12F","10"&X"F52730164E4BB843",
"11"&X"0E680199DFD9C2EB","11"&X"3C09AEEE351AA83B","11"&X"851D64EAE18AA2DB",
"11"&X"DBF524AA04D8429B","11"&X"EE9611EB2E67179B"),
("10"&X"F44F3002635124CB","11"&X"0191D8376AD381DF","11"&X"0E68007E2ABF7573",
"11"&X"26D4D1F8A85B121B","11"&X"52DA7DA86A51BF6B","11"&X"98E6202986C1DA0B",
"11"&X"E2C70067B5FFF9CB","11"&X"F1F4E188784ECECB"),
("11"&X"2394D0B6CBD97F2B","11"&X"300478E83CE2493F","11"&X"3C09A127BF17CFD3",
"11"&X"52DA7247DD7B847B","11"&X"7BC01A4CCF46A9CB","11"&X"BBEB9B6BEFF4D46B",
"11"&X"ED02319A9ED6D42B","11"&X"F6E57E69E5D4292B"),
("11"&X"6FC011D97C9C93EB","11"&X"7AA1BA0448D99FFF","11"&X"851CE23750B8B493",
"11"&X"98E5B2F9EA4E693B","11"&X"BBEB57CC25721E8B","11"&X"F116BD7BC06BA92B",
"11"&X"F525259BA7B0E8EB","11"&X"FACD84AD9F853DEB"),
("11"&X"D3C393572D94106B","11"&X"D7F13B678172087F","11"&X"DBD0637985B5D113",
"11"&X"E2A933B56BBA45BB","11"&X"ECEED51043C6DB0B","11"&X"F51A1F6ED46FA5AB",
"11"&X"6920A191848A656B","11"&X"FE904BEC7CC0BA6B"),
("11"&X"E91D93A8A9F8B66B","11"&X"EB5B3BB169963E7F","11"&X"ED6A63BAEF53F713",
"11"&X"F10333D9E6F36BBB","11"&X"F648D48ABB1A810B","11"&X"FA7419D199CA4BAB",
"11"&X"FE7922910F070B6B","11"&X"13DE3137DB55626C")
);
------------------- BM:= 2^14*M^2 ----------------------------------
constant BM : type_RNS:=(
"11"&X"FFFFBFFFFF0BC03F","11"&X"FFFFBFFFFF0C5C3B",
"11"&X"FFFFBFFFFF122837","11"&X"FFFFBFFFFF5D502F",
"11"&X"FFFFC0000391E01F","11"&X"FFFFC0004323FFFF",
"11"&X"FFFFC041008CBF3F","11"&X"FFFFC408050DBE3F"
);
--------------------------------------------------------------------
constant UDIM :type_UDIM :=(
X"8017AF3C497FF",X"8015484AC17FF",
X"80131815D97FF",X"800F51B5E97FF",
X"8009E889897FF",X"8005A73EC97FF",
X"80019106497FF",X"8000CCD0497FF"
);
------------------  M mod mi  ---------------------------------------
constant MI : type_RNS := (
"00"&X"03FFFFFEFFFFFC2F","01"&X"F3FFFFFEFFFFFC2F",
"11"&X"63FFFFFEFFFFFC41","00"&X"C3FFFFFEFFFFFD72",
"01"&X"83FFFFFF00000E3B","11"&X"03FFFFFF00010C5F",
"00"&X"03FFFFFF01040232","00"&X"03FFFFFF10201435"
);
--------------------------   -M mod mi    --------------------------
constant NI : type_RNS := (
"11"&X"FC000001000003D0","10"&X"0C000001000003CC",
"00"&X"9C000001000003B6","11"&X"3C0000010000027D",
"10"&X"7C000000FFFFF1A4","00"&X"FC000000FFFEF360",
"11"&X"FC000000FEFBFCCD","11"&X"FC000000EFDFE9CA"
);
-------------------------------------------------------------------
function Read_Moduli(i: std_logic_vector)           return   std_logic_vector;
function Read_Moduli(i: integer         )           return   std_logic_vector;
function Read_CModuli(i: std_logic_vector)          return   std_logic_vector;
function Read_CModuli(i: integer         )          return   std_logic_vector;
function Read_Dinv(I :std_logic_vector)             return   std_logic_vector;
function Read_UDIM(I: std_logic_vector)             return   std_logic_vector;
function Read_DiMmj(I:std_logic_vector)             return   type_RNS;
function BLOCK_SUM(ARNS,BRNS: in type_RNS)          return   type_RNS;
function BLOCK_SUB(ARNS,BRNS: in type_RNS)          return   type_RNS;
function BLOCK_MULTIPLICATION(ARNS,BRNS: in type_RNS)
return   type_RNS;
function Read_MDMmi(I : std_logic_vector)           return   type_RNS;
function MSBB(A: std_logic_vector )                 return   integer;
function CSA_ADDER(X,Y,Z: std_logic_vector )        return   std_logic_vector;
function BSI(X: std_logic_vector; I:integer)        return   std_logic_vector;
function ModuliMult(A: type_RNS; B: std_logic_vector)
```

```
return   type_DRNS;
function KMULT(K : std_logic_vector)                return   type_RNS;
function BLOCK_3SUM(ARNS,BRNS,CRNS: in type_RNS)    return   type_RNS;

procedure MODULI_ADD(
signal    A :     IN  std_logic_vector(w-1 downto 0);
signal    B :     IN  std_logic_vector(w-1 downto 0);
signal    I :     IN  std_logic_vector(c-1 downto 0);
signal    ARNS :  OUT std_logic_vector(w-1 downto 0)
                     );
------------------------------------------------------------------
procedure MODULI_SUB(
signal    A :     IN  std_logic_vector(w-1 downto 0);
signal    B :     IN  std_logic_vector(w-1 downto 0);
signal    I :     IN  std_logic_vector(c-1 downto 0);
signal    ARNS :  OUT std_logic_vector(w-1 downto 0)
                    );

------------------------------------------------------------------
procedure KOM(
signal    A :     IN  std_logic_vector(w-1 downto 0);
signal    B :     IN  std_logic_vector(w-1 downto 0);
signal    C :     OUT std_logic_vector(2*w-1 downto 0)
             );
------------------------------------------------------------------
procedure BRTR(
             signal A: in  std_logic_vector(2*w-1 downto 0);
             signal I: in  integer;
             signal R: out std_logic_vector(w-1 downto 0)
             );

procedure BRTR(
             signal A: in  std_logic_vector(2*w-1 downto 0);
             signal I: in  std_logic_vector(3   downto 0);
             signal R: out std_logic_vector(w-1 downto 0)
             );
procedure MMR(
             signal A: in  std_logic_vector(2*w-1 downto 0);
             signal I: in  std_logic_vector(3 downto 0);
             signal R: out std_logic_vector(w-1 downto 0)
             );
procedure MMR(
             signal A: in  std_logic_vector(2*w-1 downto 0);
                    I: in  integer;
             signal R: out std_logic_vector(w-1 downto 0)
              );
---------------------------------------------------------------------------
procedure MMR_M1(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M2(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M3(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M4(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M5(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M6(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M7(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
procedure MMR_M8(
              signal A: in  std_logic_vector(2*w-1 downto 0);
              signal R: out std_logic_vector(w-1 downto 0));
------------------------------------------------------------------------
procedure YRNS(
              signal K : IN  type_YACU;
              signal XI: OUT type_RNS);
------------------------------------------------------------------------
end package;
package body RNS_PACKAGE is

-------------------------------------------------------
function Read_Moduli(i: std_logic_vector)
return std_logic_vector is
variable j: integer range 1 to total_channels;
begin
if I >=1 and I<= total_channels then
   j:=conv_integer(I);
else
   j:=1;
end if;
return Moduli(j);
end function Read_Moduli;
------------------------------------------------------
function Read_CModuli(i: std_logic_vector)
return std_logic_vector is
variable j: integer range 1 to total_channels;
begin
if I >=1 and I<= total_channels then
   j:=conv_integer(I);
else
   j:=1;
end if;
return CModuli(j);
end function Read_CModuli;
--------------------------------------------------------
function Read_Moduli(i: integer)
return std_logic_vector is
variable j: integer range 1 to total_channels;
begin
if I >=1 and I<= total_channels then
   j:=I;
else
   j:=1;
end if;
return Moduli(j);
end function Read_Moduli;
----------------------------------------------------------
```

```
function Read_CModuli(i: integer)
return std_logic_vector is
variable j: integer range 1 to total_channels;
begin
if I >=1 and I<= total_channels then
   j:=I;
else
   j:=1;
end if;
return CModuli(j);
end function Read_CModuli;
--------------------------------------------------------------------------
function Read_Dinv(I :std_logic_vector)
return std_logic_vector is
variable j: integer range 1 to total_channels;
begin
if I >=1 and I<=total_channels then
   j:=conv_integer(I);
else
   j:=1;
end if;
return DINV(j);
end function Read_Dinv;
--------------------------------------------------------------------------
function Read_DiMmj(I:std_logic_vector)
return type_RNS is
--variable Rom_DiMmj: type_RNS_Table;
variable j: integer range 1 to total_channels;
begin
if I >=1 and I<=total_channels then
   j:=conv_integer(I);
else
   j:=1;
end if;
return DiMmj(j);
end function Read_DiMmj;
--------------------------------------------------------------------------
function Read_MDMmi(I : std_logic_vector) return type_RNS is
variable Rom_AlphaDMm : type_RNS_Table;
variable j: integer range 1 to total_channels-1;
begin
Rom_AlphaDMm(1)  :=("00"&X"9DB5B938A5353587","01"&X"6B9B4BF62062D19C",
"01"&X"CC1C6014E1C79339","01"&X"5A09201E34488F0B","00"&X"3D3588F803A6A0EE",
"01"&X"BE1E4A87A7C4ECD0","00"&X"E35509A9308955D3","10"&X"9463CB9AEE1D079A");
Rom_AlphaDMm(2)  :=("01"&X"3B6B72714A6A6B0E","10"&X"D73697EC40C5A338",
"11"&X"9838C029C38F2672","10"&X"B412403C68911E16","00"&X"7A6B11F0074D41DC",
"11"&X"7C3C950F4F89D9A0","01"&X"C6AA13526112ABA6","01"&X"28C79735DC3A1135");
Rom_AlphaDMm(3)  :=("01"&X"D9212BA9EF9FA095","00"&X"42D1E3E2612874D9",
"01"&X"6455203EA556B9B4","00"&X"0E1B605A9CD9AD32","00"&X"B7A09AE80AF3E2CA",
"01"&X"3A5ADF96F74EC6B1","10"&X"A9FF1CFB919C0179","11"&X"BD2B62D0CA5718CF");

Rom_AlphaDMm(4)  :=("10"&X"76D6E4E294D4D61C","01"&X"AE6D2FD8818B4675",
"11"&X"30718053871E4CED","01"&X"68248078D1223C3D","00"&X"F4D623E00E9A83B8",
"10"&X"F8792A1E9F13B381","11"&X"8D5426A4C225574C","10"&X"518F2E6BB874226A");
Rom_AlphaDMm(5)  :=("11"&X"148C9E1B3A0A0BA3","11"&X"1A087BCEA1EE1811",
"00"&X"FC8DE06868E5E02F","10"&X"C22DA097056ACB48","01"&X"320BACD8124124A6",
"00"&X"B69774A646D8A092","00"&X"70A9304DF2AEAE20","00"&X"E5F2FA06A6912C05");
```

```
Rom_AlphaDMm(6)  :=("11"&X"B2425753DF3F412A","00"&X"85A3C7C4C250E9B2",
"10"&X"C8AA407D4AAD7368","00"&X"1C36C0B539B35A64","01"&X"6F4135D015E7C594",
"10"&X"74B5BF2DEE9D8D62","01"&X"53FE39F7233803F3","11"&X"7A56C5A194AE339F");
Rom_AlphaDMm(7)  :=("00"&X"4FF8108C847476B2","01"&X"F13F13BAE2B3BB4E",
"00"&X"94C6A0922C7506AA","01"&X"763FE0D36DFBE96F","01"&X"AC76BEC8198E6682",
"00"&X"32D409B596627A73","10"&X"375343A053C159C6","10"&X"0EBA913C82CB3D3A");

if I >= 1 and I<= total_channels-1 then
   j:=conv_integer(I);
else
   j:=1;
end if;
return Rom_AlphaDMm(j);
end function Read_MDMmi;
-------------------------------------------------------------------------
function Read_UDIM(I: std_logic_vector)
return   std_logic_vector is
variable j: integer range 1 to total_channels;
begin
if I >=1 and I<=total_channels then
   j:=conv_integer(I);
else
   j:=1;
end if;
return UDIM(j);
end function Read_UDIM;
-------------------------------------------------------------------------
function BLOCK_SUM(ARNS,BRNS: in type_RNS)
return type_RNS is
variable j: integer;
variable C: type_RNS;
variable Sum,S,A,B: std_logic_vector(w downto 0):=(others=>'0');
begin
for j in 1 to total_channels loop
A:=('0'& ARNS(j));
B:=('0'& BRNS(j));
Sum:= A+B;
if Sum >= Moduli(j) then
   S:= Sum - Moduli(j);
else
   S:= Sum;
end if;
C(j):= S(w-1 downto 0);
end loop;
return C;
end function;
-------------------------------------------------------------------------
function BLOCK_3SUM(ARNS,BRNS,CRNS: in type_RNS)
return type_RNS is
variable j: integer;
variable DRNS     : type_RNS;
variable A,B,N,S1    : std_logic_vector(w downto 0);
variable C           : std_logic_vector(w downto 0);
variable X,Y      : std_logic_vector(w+1 downto 0);
variable S,SUM    : std_logic_vector(w+1 downto 0);
begin

for j in 1 to total_channels loop
```

```
N:= (Moduli(j))&'0';
A:=('0'& ARNS(j));
B:=('0'& BRNS(j));
C:=('0'& CRNS(j));
S1(w downto 0):= A(w downto 0)+B(w downto 0);
SUM(w+1 downto 0) := '0'&S1(w downto 0)+ C;

if (Sum >= N) then
   S:= Sum - N;
elsif (SUM > Moduli(j)) then
   S:= Sum - Moduli(j);
else
   S:= Sum;
end if;
DRNS(j):= S(w-1 downto 0);
end loop;
return DRNS;
end function;
-------------------------------------------------------------------------
function BLOCK_SUB(ARNS,BRNS: in type_RNS)
return type_RNS is
variable j: integer;
variable C: type_RNS;
variable Sub,S,A,B: std_logic_vector(w downto 0);
variable E: std_logic_vector(w-1 downto 0);
begin
for j in 1 to total_channels loop
A:=('0'& ARNS(j));
B:=('0'& BRNS(j));
S:= A + BM(j);
Sub := S - B;
if Sub > Moduli(j) then
   Sub(w downto 0):= Sub(w downto 0)+ CModuli(j);
end if;
C(j):= Sub(w-1 downto 0);
end loop;
return C;
end function;
------------------------------------------------------------------------
function BLOCK_MULTIPLICATION(ARNS,BRNS: in type_RNS)
return type_RNS is
variable i: integer;
variable Ci: type_RNS;
variable A: std_logic_vector(2*w-1 downto 0);
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM,N,Z : std_logic_vector(w downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(47 downto 0):=(others=>'0');
begin
for i in 1 to total_channels loop
A:= ARNS(i)*BRNS(i);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
M:= Moduli(i);
case i is
   when 2 =>
       AAL:= AH(w-3 downto 0)&"00";
```

```
        AAH:= "00000000000000"&Z0&AH(w-1 downto w-2)&AH(w-1 downto w-2);
    when 3 =>
        AAL:= AH(w-4 downto 0)&"000";
        AAH:= "000000000000"&Z0&AH(w-1 downto w-3)&AH(w-1 downto w-3);
    when 4 =>
        AAL:= AH(w-5 downto 0)&"0000";
        AAH:= "0000000000"&Z0&AH(w-1 downto w-4)&AH(w-1 downto w-4);
    when 5 =>
        AAL:= AH(w-6 downto 0)&"00000";
        AAH:= "00000000"&Z0&AH(w-1 downto w-5)&AH(w-1 downto w-5);
    when 6 =>
        AAL:= AH(w-7 downto 0)&"000000";
        AAH:= "000000"&Z0&AH(w-1 downto w-6)&AH(w-1 downto w-6);
    when 7 =>
        AAL:= AH(w-9 downto 0)&"00000000";
        AAH:= "00"&Z0&AH(w-1 downto w-8)&AH(w-1 downto w-8);
    when 8 =>
        AAL:= AH(w-10 downto 0)&"000000000";
        AAH:= Z0&AH(w-1 downto w-9)&AH(w-1 downto w-9);
    when others =>
        AAL:= (others=>'0');
        AAH:= (others=>'0');
    end case;
    N:= AAL+AAH+AM;
  if (N>= M) then
     Z(w downto 0):= N(w downto 0)-M(w-1 downto 0);
  else
     Z(w downto 0):= N(w downto 0);
  end if;
  Ci(i) := Z(w-1 downto 0);
end loop;
return Ci;
end function BLOCK_MULTIPLICATION;
-------------------------------------------------------------------------
function ModuliMult(A: type_RNS; B: std_logic_vector)
return   type_DRNS is
variable V : type_DRNS;
begin
for i in 1 to total_channels loop
V(i):= A(i)*B;
end loop;
return V;
end function ModuliMult;

function KMULT(K : std_logic_vector)
return type_RNS is
variable Ki: type_RNS;
variable A: std_logic_vector(2*w-15 downto 0);
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM,N,Z : std_logic_vector(w downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(47 downto 0):=(others=>'0');
begin
for i in 1 to total_channels loop
A:= K(w-15 downto 0)*NI(i);
AH:= "00000000000000"&A(2*w-15 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
```

```
M:= Moduli(i);
case i is
   when 2 =>
       AAL:= AH(w-3 downto 0)&"00";
       AAH:= "00000000000000"&Z0&AH(w-1 downto w-2)&AH(w-1 downto w-2);
   when 3 =>
       AAL:= AH(w-4 downto 0)&"000";
       AAH:= "000000000000"&Z0&AH(w-1 downto w-3)&AH(w-1 downto w-3);
   when 4 =>
       AAL:= AH(w-5 downto 0)&"0000";
       AAH:= "0000000000"&Z0&AH(w-1 downto w-4)&AH(w-1 downto w-4);
   when 5 =>
       AAL:= AH(w-6 downto 0)&"00000";
       AAH:= "00000000"&Z0&AH(w-1 downto w-5)&AH(w-1 downto w-5);
   when 6 =>
       AAL:= AH(w-7 downto 0)&"000000";
       AAH:= "000000"&Z0&AH(w-1 downto w-6)&AH(w-1 downto w-6);
   when 7 =>
       AAL:= AH(w-9 downto 0)&"00000000";
       AAH:= "00"&Z0&AH(w-1 downto w-8)&AH(w-1 downto w-8);
   when 8 =>
       AAL:= AH(w-10 downto 0)&"000000000";
       AAH:= Z0&AH(w-1 downto w-9)&AH(w-1 downto w-9);
   when others =>
       AAL:= (others=>'0');
       AAH:= (others=>'0');
   end case;
   N:= AAL+AAH+AM;
 if (N>= M) then
    Z(w downto 0):= N(w downto 0)-M(w-1 downto 0);
 else
    Z(w downto 0):= N(w downto 0);
 end if;
  Ki(i) := Z(w-1 downto 0);
end loop;
return Ki;
end function KMULT;
---------------------------------------------------------------------------
function MSBB(A: std_logic_vector ) return integer is
  variable n : integer;
  begin
  for i in A'range loop
     if A(i)='1' then
     n:= i;
     exit;
     end if;
  end loop;
  return n;
end function MSBB;
---------------------------------------------------------------------------
function CSA_ADDER(X,Y,Z: std_logic_vector)
return std_logic_vector is
variable S,C,D: std_logic_vector(2*w downto 0);
begin
S := '0' & ((X(2*w-1 downto 0) XOR Y(2*w-1 downto 0)) XOR Z(2*w-1 downto 0));
C := (((X(2*w-1 downto 0) AND Y(2*w-1 downto 0)) OR (X(2*w-1 downto 0) AND
Z(2*w-1 downto 0))) OR (Y(2*w-1 downto 0) AND Z(2*w-1 downto 0)))& '0';
D := S+C;
```

```
return D;
end function CSA_ADDER;
---------------------------------------------------------------------
function BSI(X:std_logic_vector; I: integer)  return   std_logic_vector is
variable Z0: std_logic_vector(w-10 downto 0):= (others=>'0');
variable Z1: std_logic_vector(w+8 downto 0);
variable Z : std_logic_vector(2*w-1 downto 0);
begin
Z(2*w-1 downto w+9) := (others =>'0');
if  i = 2 then
Z(w+1  downto 2) := X(w-1 downto 0);
Z(1 downto 0) := (others=>'0');
Z(w+8 downto w+2) := (others=>'0');
elsif i = 3 then
Z(w+2  downto 3) := X(w-1 downto 0);
Z(2 downto 0) := (others=>'0');
Z(w+8 downto w+3) := (others=>'0');
elsif i = 4 then
Z(w+3  downto 4) := X(w-1 downto 0);
Z(3 downto 0) := (others=>'0');
Z(w+8 downto w+4) := (others=>'0');
elsif i = 5 then
Z(w+4  downto 5) := X(w-1 downto 0);
Z(4 downto 0) := (others=>'0');
Z(w+8 downto w+5) := (others=>'0');
elsif i = 6 then
Z(w+5  downto 6) := X(w-1 downto 0);
Z(5 downto 0) := (others=>'0');
Z(w+8 downto w+6) := (others=>'0');
elsif i = 7 then
Z(w+6  downto 7) := X(w-1 downto 0);
Z(7 downto 0) := (others=>'0');
Z(w+8 downto w+7) := (others=>'0');
elsif i = 8 then
Z(w+8 downto 9) := X(w-1 downto 0);
Z(8 downto 0) := (others=>'0');
else
Z(w+8 downto 0):=(others=>'0');
end if;
return Z;
end function BSI;
--------------------------------------------------------------------
procedure MODULI_ADD(
signal  A :     IN  std_logic_vector(w-1 downto 0);
signal  B :     IN  std_logic_vector(w-1 downto 0);
signal  I :     IN  std_logic_vector(c-1 downto 0);
signal  ARNS :  OUT std_logic_vector(w-1 downto 0)) is
Variable SUM1: std_logic_vector(w downto 0);
Variable SUM2: std_logic_vector(w downto 0);
Variable Mi  : std_logic_vector(w-1 downto 0);
begin
Mi(w-1 downto 0) := Read_CModuli(I);
SUM2(w downto 0) := '0'&A + B;
SUM1(w downto 0) := '0'&A + B + Mi;
if SUM1(w)='1' then
   ARNS(w-1 downto 0) <= SUM1(w-1 downto 0);
else
   ARNS(w-1 downto 0) <= SUM2(w-1 downto 0);
```

```
end if;
end procedure MODULI_ADD;
-------------------------------------------------------------------------

procedure MODULI_SUB(
signal  A :     IN  std_logic_vector(w-1 downto 0);
signal  B :     IN  std_logic_vector(w-1 downto 0);
signal  I :     IN  std_logic_vector(c-1 downto 0);
signal  ARNS :  OUT std_logic_vector(w-1 downto 0)) is
Variable Mi  : std_logic_vector(w-1 downto 0);
begin
Mi(w-1 downto 0) := Read_Moduli(I);

if A >= B then
   ARNS(w-1 downto 0) <= A-B;
else
   ARNS(w-1 downto 0) <= A+Mi-B;
end if;
end procedure MODULI_SUB;
-------------------------------------------------------------------------
procedure KOM(
signal    A :     IN  std_logic_vector(w-1 downto 0);
signal    B :     IN  std_logic_vector(w-1 downto 0);
signal    C :     OUT std_logic_vector(2*w-1 downto 0)
             ) is
variable X0,Y0,X1,Y1:     std_logic_vector(w/2-1 downto 0);
variable XX,YY:           std_logic_vector(w/2-1 downto 0);
variable XY0,XY1:         std_logic_vector(w-1   downto 0);
variable XY2:             std_logic_vector(w     downto 0);
variable XY:              std_logic_vector(w-1   downto 0);
variable XY3:             std_logic_vector(w+1   downto 0);

variable LXY0,LXY1,LXY2:  std_logic_vector(2*w-1 downto 0);
variable Z1 :             std_logic_vector(w-1   downto 0) := (others=>'0');
variable Z2 :             std_logic_vector(w/2-1 downto 0) := (others=>'0');
variable Z3 :             std_logic_vector(w/2-3 downto 0) := (others=>'0');
variable CC :             std_logic_vector(2*w   downto 0);
begin
X0(w/2-1 downto 0) := A(w/2-1 downto 0);
Y0(w/2-1 downto 0) := B(w/2-1 downto 0);
X1(w/2-1 downto 0) := A(w-1 downto w/2);
Y1(w/2-1 downto 0) := B(w-1 downto w/2);

XX(w/2-1 downto 0)  := (X1(w/2-1 downto 0) - X0(w/2-1 downto 0));
YY(w/2-1 downto 0)  := (Y0(w/2-1 downto 0) - Y1(w/2-1 downto 0));
XY0(w-1 downto 0)   := X0(w/2-1 downto 0)* Y0(w/2-1 downto 0); -- 66 bits
XY1(w-1 downto 0)   := X1(w/2-1 downto 0)* Y1(w/2-1 downto 0); -- 66 bits
XY(w-1 downto 0)    := XX(w/2-1 downto 0)* YY(w/2-1 downto 0);
XY2(w downto 0)     := '0'&XY0(w-1 downto 0)+XY1(w-1 downto 0); -- 67 bits
XY3(w+1 downto 0)   := '0'&XY2(w downto 0) + XY(w-1 downto 0);
LXY0 := (Z1&XY0);
LXY1 := ((Z3&XY3)&Z2);
LXY2 := (XY1&Z1);
CC := CSA_ADDER(LXY0,LXY1,LXY2);
C  <=CC(2*w-1 downto 0);
end procedure KOM;

procedure BRTR(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal I: in  integer;
signal R: out std_logic_vector(w-1 downto 0)) is

variable S,S2: std_logic_vector(2*w downto 0);
variable TL,TH,SL,SH,TM,SM,RZ : std_logic_vector(2*w-1 downto 0);
variable A2,A3,A4,A5,A6,A7,A8   : std_logic_vector(2*w-1 downto 0);
variable B2,B3,B4,B5,B6,B7,B8   : std_logic_vector(2*w-1 downto 0);
constant Z :  std_logic_vector(w-1  downto 0):=(others=>'0');
constant Z0:  std_logic_vector(w-10 downto 0):=(others=>'0');

begin
A2:= "0000000"&Z0&A(2*w-1 downto w)&"00";
A3:=  "000000"&Z0&A(2*w-1 downto w)&"000";
A4:=   "00000"&Z0&A(2*w-1 downto w)&"0000";
A5:=    "0000"&Z0&A(2*w-1 downto w)&"00000";
A6:=     "000"&Z0&A(2*w-1 downto w)&"000000";
A7:=       '0'&Z0&A(2*w-1 downto w)&"00000000";
A8:=            Z0&A(2*w-1 downto w)&"000000000";
TH:=  Z& A(2*w-1 downto w);
TM := A(2*w-1 downto w)& Z;
if i=2 then
   TL:=A2;
elsif i=3 then
   TL:=A3;
elsif i=4 then
   TL:=A4;
elsif i=5 then
   TL:=A5;
elsif i=6 then
   TL:=A6;
elsif i=7 then
   TL:=A7;
elsif i=8 then
   TL:=A8;
else
   TL:=(others=>'0');
end if;
S := CSA_ADDER(TH,TM,TL);
SM:= S(2*w-1 downto w) &Z;
SH:= Z &S(2*w-1 downto w);

B2:= "0000000"&Z0&S(2*w-1 downto w)&"00";
B3:=  "000000"&Z0&S(2*w-1 downto w)&"000";
B4:=   "00000"&Z0&S(2*w-1 downto w)&"0000";
B5:=    "0000"&Z0&S(2*w-1 downto w)&"00000";
B6:=     "000"&Z0&S(2*w-1 downto w)&"000000";
B7:=       '0'&Z0&S(2*w-1 downto w)&"00000000";
B8:=            Z0&S(2*w-1 downto w)&"000000000";

if i=2 then
   SL:=B2;
elsif i=3 then
   SL:=B3;
elsif i=4 then
   SL:=B4;
elsif i=5 then
   SL:=B5;
```

```
elsif i=6 then
   SL:=B6;
elsif i=7 then
   SL:=B7;
elsif i=8 then
   SL:=B8;
else
   SL:=(others=>'0');
end if;
S2 := CSA_ADDER(A,SL,SH);
RZ(2*w-1 downto 0) := S2(2*w-1 downto 0)-SM(2*w-1 downto 0);
R <= RZ(w-1 downto 0);
end procedure BRTR;

procedure BRTR(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal I: in  std_logic_vector(3 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is

variable S,S2: std_logic_vector(2*w downto 0);
variable TL,TH,SL,SH,TM,SM,RZ : std_logic_vector(2*w-1 downto 0);
variable A2,A3,A4,A5,A6,A7,A8    : std_logic_vector(2*w-1 downto 0);
variable B2,B3,B4,B5,B6,B7,B8    : std_logic_vector(2*w-1 downto 0);
constant Z :  std_logic_vector(w-1  downto 0):=(others=>'0');
constant Z0:  std_logic_vector(w-10 downto 0):=(others=>'0');

begin
A2:= "0000000"&Z0&A(2*w-1 downto w)&"00";
A3:=  "000000"&Z0&A(2*w-1 downto w)&"000";
A4:=   "00000"&Z0&A(2*w-1 downto w)&"0000";
A5:=    "0000"&Z0&A(2*w-1 downto w)&"00000";
A6:=     "000"&Z0&A(2*w-1 downto w)&"000000";
A7:=       '0'&Z0&A(2*w-1 downto w)&"00000000";
A8:=           Z0&A(2*w-1 downto w)&"000000000";
TH:=  Z& A(2*w-1 downto w);
TM := A(2*w-1 downto w)& Z;
if i="0010" then
   TL:=A2;
elsif i="0011" then
   TL:=A3;
elsif i="0100" then
   TL:=A4;
elsif i="0101" then
   TL:=A5;
elsif i="0110" then
   TL:=A6;
elsif i="0111" then
   TL:=A7;
elsif i="1000" then
   TL:=A8;
else
   TL:=(others=>'0');
end if;
S := CSA_ADDER(TH,TM,TL);
SM:= S(2*w-1 downto w) &Z;
SH:= Z &S(2*w-1 downto w);

B2:= "0000000"&Z0&S(2*w-1 downto w)&"00";
```

```
B3:=  "000000"&Z0&S(2*w-1 downto w)&"000";
B4:=   "00000"&Z0&S(2*w-1 downto w)&"0000";
B5:=    "0000"&Z0&S(2*w-1 downto w)&"00000";
B6:=     "000"&Z0&S(2*w-1 downto w)&"000000";
B7:=       '0'&Z0&S(2*w-1 downto w)&"00000000";
B8:=            Z0&S(2*w-1 downto w)&"000000000";

if i="0010" then
   SL:=B2;
elsif i="0011" then
   SL:=B3;
elsif i="0100" then
   SL:=B4;
elsif i="0101" then
   SL:=B5;
elsif i="0110" then
   SL:=B6;
elsif i="0111" then
   SL:=B7;
elsif i="1000" then
   SL:=B8;
else
   SL:=(others=>'0');
end if;
S2 := CSA_ADDER(A,SL,SH);
RZ(2*w-1 downto 0) := S2(2*w-1 downto 0)-SM(2*w-1 downto 0);
if( RZ < Read_Moduli(I)) then
R <= RZ(w-1 downto 0);
else
R <= RZ(w-1 downto 0)- READ_MODULI(I);
end if;

end procedure BRTR;


procedure MMR(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal I: in  std_logic_vector(3 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM     : std_logic_vector(w downto 0);
variable N      : std_logic_vector(w+1 downto 0);
variable M      : std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(47 downto 0):=(others=>'0');
begin
M:= Read_Moduli(I);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM-M;
end if;
case I is
   when "0010" =>
        AAL:= AH(w-3 downto 0)&"00";
        AAH:= "00000000000000"&Z0&AH(w-1 downto w-2)&AH(w-1 downto w-2);
   when "0011" =>
        AAL:= AH(w-4 downto 0)&"000";
```

```
       AAH:= "000000000000"&Z0&AH(w-1 downto w-3)&AH(w-1 downto w-3);
   when "0100" =>
       AAL:= AH(w-5 downto 0)&"0000";
       AAH:= "0000000000"&Z0&AH(w-1 downto w-4)&AH(w-1 downto w-4);
   when "0101" =>
       AAL:= AH(w-6 downto 0)&"00000";
       AAH:= "00000000"&Z0&AH(w-1 downto w-5)&AH(w-1 downto w-5);
   when "0110" =>
       AAL:= AH(w-7 downto 0)&"000000";
       AAH:= "000000"&Z0&AH(w-1 downto w-6)&AH(w-1 downto w-6);
   when "0111" =>
       AAL:= AH(w-9 downto 0)&"00000000";
       AAH:= "00"&Z0&AH(w-1 downto w-8)&AH(w-1 downto w-8);
   when "1000" =>
       AAL:= AH(w-10 downto 0)&"000000000";
       AAH:= Z0&AH(w-1 downto w-9)&AH(w-1 downto w-9);
   when others =>
       AAL:= (others=>'0');
       AAH:= (others=>'0');
   end case;
   N:= '0'&AM+AAL+AAH;
 if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
 end if;
  R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR;

procedure MMR(
signal A: in  std_logic_vector(2*w-1 downto 0);
       i: in  integer;
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM    : std_logic_vector(w downto 0);
variable N     : std_logic_vector(w+1 downto 0);
variable M     :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(47 downto 0):=(others=>'0');
begin
M:= Moduli(i);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM-M;
end if;
case i is
   when 2 =>
       AAL:= AH(w-3 downto 0)&"00";
       AAH:= "00000000000000"&Z0&AH(w-1 downto w-2)&AH(w-1 downto w-2);
   when 3 =>
       AAL:= AH(w-4 downto 0)&"000";
       AAH:= "000000000000"&Z0&AH(w-1 downto w-3)&AH(w-1 downto w-3);
   when 4 =>
       AAL:= AH(w-5 downto 0)&"0000";
       AAH:= "0000000000"&Z0&AH(w-1 downto w-4)&AH(w-1 downto w-4);
   when 5 =>
       AAL:= AH(w-6 downto 0)&"00000";
       AAH:= "00000000"&Z0&AH(w-1 downto w-5)&AH(w-1 downto w-5);
   when 6 =>
```

```
       AAL:= AH(w-7 downto 0)&"000000";
       AAH:= "000000"&Z0&AH(w-1 downto w-6)&AH(w-1 downto w-6);
   when 7 =>
       AAL:= AH(w-9 downto 0)&"00000000";
       AAH:= "00"&Z0&AH(w-1 downto w-8)&AH(w-1 downto w-8);
   when 8 =>
       AAL:= AH(w-10 downto 0)&"000000000";
       AAH:= Z0&AH(w-1 downto w-9)&AH(w-1 downto w-9);
   when others =>
       AAL:= (others=>'0');
       AAH:= (others=>'0');
   end case;
 N:= '0'&AM+AAL+AAH;
 if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
 end if;
  R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR;
-------------------------------------------------------------------------
procedure MMR_M1(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH : std_logic_vector(w-1 downto 0);
variable N     : std_logic_vector(w downto 0);
variable M     : std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(47 downto 0):=(others=>'0');
begin
M:= Moduli(1);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
N:= '0'&AH+AL;
 if (N>= M) then
    N(w downto 0):= N(w downto 0)-M(w-1 downto 0);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M1;
-------------------------------------------------------------------------
procedure MMR_M2(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM  : std_logic_vector(w downto 0);
variable N,Z : std_logic_vector(w+1 downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(61 downto 0):=(others=>'0');
begin
M:= Moduli(2);
AH(w-1 downto 0) := A(2*w-1 downto w);
AL(w-1 downto 0) := A(w-1 downto 0);
AAL(w-1 downto 0):= A(2*w-3 downto w)&"00";
AAH(w-1 downto 0):= Z0&(A(2*w-1 downto 2*w-2)&A(2*w-1 downto 2*w-2));
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM - M;
end if;
N:= '0'&AM+AAL+AAH;
 if (N>=Moduli(2)) then
    N(w+1 downto 0):= N(w+1 downto 0)-moduli(2);
 else
    N(w+1 downto 0):= N(w+1 downto 0);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M2;

------------------------------------------------------------------------
procedure MMR_M3(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM : std_logic_vector(w downto 0);
variable N  : std_logic_vector(w+1 downto 0);
variable M    :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(59 downto 0):=(others=>'0');
begin
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
M:= Moduli(3);
if AM >= M then
   AM:= AM - M;
end if;
AAL:= AH(w-4 downto 0)&"000";
AAH:= Z0&AH(w-1 downto w-3)&AH(w-1 downto w-3);
N:= '0'&AM+ AAH+AAL;
 if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-Moduli(3);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M3;
-----------------------------------------------------------------
procedure MMR_M4(
signal A     :  in  std_logic_vector(2*w-1 downto 0);
signal R     :  out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM  :  std_logic_vector(w   downto 0);
variable N   :  std_logic_vector(w+1 downto 0);
variable M   :  std_logic_vector(w-1 downto 0);
constant Z0  :  std_logic_vector(w-9 downto 0):=(others=>'0');
begin
M:= Moduli(4);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM - M;
end if;
AAL:= A(2*w-5 downto w)&"0000";
AAH:= Z0&A(2*w-1 downto 2*w-4)&A(2*w-1 downto 2*w-4);
N(w+1 downto 0) := '0'&AM+AAL+AAH;
 if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M4;
-----------------------------------------------------------------
procedure MMR_M5(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM: std_logic_vector(w downto 0);
variable N : std_logic_vector(w+1 downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(w-11 downto 0):=(others=>'0');
begin
M:= Moduli(5);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM - M;
end if;
AAL:= A(2*w-6 downto w)&"00000";
AAH:= Z0&A(2*w-1 downto 2*w-5)&A(2*w-1 downto 2*w-5);
N:= '0'&AM+AAL+AAH;
 if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
 else
    N(w+1 downto 0):= N(w+1 downto 0);
 end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M5;
-------------------------------------------------------------------
procedure MMR_M6(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM  : std_logic_vector(w downto 0);
variable N   : std_logic_vector(w+1 downto 0);
variable M   : std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(w-13 downto 0):=(others=>'0');
begin
M:= Moduli(6);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM:= AM-M;
end if;
AAL:= A(2*w-7 downto w)&"000000";
AAH:= Z0&A(2*w-1 downto 2*w-6)&A(2*w-1 downto 2*w-6);
N:= '0'&AM+AAL+AAH;
if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M6;
----------------------------------------------------------------------
procedure MMR_M7(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM  : std_logic_vector(w downto 0);
variable N   : std_logic_vector(w+1 downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
```

```
constant Z0:  std_logic_vector(w-17 downto 0):=(others=>'0');
begin
M := Moduli(7);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM := AM-M;
end if;
AAL:= A(2*w-9 downto w)&"00000000";
AAH:= Z0&A(2*w-1 downto 2*w-8)&A(2*w-1 downto 2*w-8);
N:= '0'&AM+AAL+AAH;
if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M7;
-----------------------------------------------------------------
procedure MMR_M8(
signal A: in  std_logic_vector(2*w-1 downto 0);
signal R: out std_logic_vector(w-1 downto 0)) is
variable AL,AH,AAL,AAH : std_logic_vector(w-1 downto 0);
variable AM  : std_logic_vector(w downto 0);
variable N   : std_logic_vector(w+1 downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
constant Z0:  std_logic_vector(w-19 downto 0):=(others=>'0');
begin
M:= Moduli(8);
AH:= A(2*w-1 downto w);
AL:= A(w-1 downto 0);
AM:= '0'&AH+AL;
if AM >= M then
   AM := AM-M;
end if;
AAL:= A(2*w-10 downto w)&"000000000";
AAH:= Z0&A(2*w-1 downto 2*w-9)&A(2*w-1 downto 2*w-9);
N:= '0'&AM+AAL+AAH;
if (N>= M) then
    N(w+1 downto 0):= N(w+1 downto 0)-M(w-1 downto 0);
end if;
R(w-1 downto 0) <= N(w-1 downto 0);
end procedure MMR_M8;

procedure YRNS (
signal K : IN  type_YACU;
signal XI: OUT type_RNS)  is
variable Ki    : std_logic_vector(w+2 downto 0);
variable AL    : std_logic_vector(w-1 downto 0);
variable AH    : std_logic_vector(2 downto 0);
variable AAL : std_logic_vector(11 downto 0);
variable N,Z : std_logic_vector(w downto 0);
variable AM  : std_logic_vector(w downto 0);
variable M   :  std_logic_vector(w-1  downto 0);
begin
for i in 1 to total_channels loop
   M:= read_moduli(I);
   Ki := K(i);
   AH:= Ki(w+2 downto w);
   AL:= Ki(w-1 downto 0);
   AM:= '0'&AL + AH;
   case I is
      when 2 =>
          AAL:=   "0000000"& Ki(w+2 downto w)&"00";
      when 3 =>
          AAL:=    "000000"& Ki(w+2 downto w)&"000";
      when 4 =>
          AAL:=     "00000"& Ki(w+2 downto w)&"0000";
      when 5 =>
          AAL:=      "0000"& Ki(w+2 downto w)&"00000";
      when 6 =>
          AAL:=       "000"& Ki(w+2 downto w)&"000000";
      when 7 =>
          AAL:=         "0"& Ki(w+2 downto w)&"00000000";
      when 8 =>
          AAL:=              Ki(w+2 downto w)&"000000000";
      when others =>
          AAL:= (others=>'0');
      end case;
      N:= AM + AAL;
    if (N>= M) then
       Z(w downto 0):= N(w downto 0)-M(w-1 downto 0);
    else
       Z(w downto 0):= N(w downto 0);
    end if;
    XI(i) <= Z(w-1 downto 0);
end loop;
end procedure;

end package body RNS_PACKAGE;
```

B.4 Hardware description for SOR_2 algorithm

```
--------------------------------------------------
-- Written by: Ali Mehrabi
-- Create Date: 27.06.2018 12:46:57
-- Design Name:
-- Module Name: SOMR - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-------------------------------------------------
library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.RNS_package.all;
```

```
entity SMR is
Port (
      CLK   : IN std_logic;
      RESETN: IN std_logic;
      Ai    : IN type_RNS;
      Zi    : OUT type_RNS
      );
end SMR;
architecture Behavioral of SMR is
type     type_STATE is (T01,T02,T03,T04,T05,T06,T07,T08,T09,T10,T11,T12,
T13,T14,T15,T16,T17,T18,T19,T20,T21,T22);
type     type_ST    is (S01,S02,S03,S04,S05,S06,S07,S08,S09,S10,S11,S12,
S13,S14,S15,S16,S17);
constant DELTA              :  std_logic_vector(4 downto 0) :="10000";
signal   STATE              :  type_STATE;
signal   ST                 :  type_ST;
signal   G1,G2              :  std_logic_vector(w-1   downto 0);
signal   Alpha              :  std_logic_vector(c-2   downto 0);
signal   ENY,CLKY           :  std_logic;
signal   AL,ALP             :  std_logic_vector(10     downto 0);
signal   K1_REG,K2_REG,K1,K2 :  std_logic_vector(2*w-15 downto 0);  -- 118 bits
signal   K_SUM              :  std_logic_vector(2*w-14 downto 0);  -- 132 bits
signal   K_ACCU             :  std_logic_vector(2*w-11 downto 0);  -- 121 bits
--signal   K                  :  std_logic_vector(w-15   downto 0);  --just for test
signal   UD1,UD2            :  std_logic_vector(w-15    downto 0);
signal   AlphaD,KM          :  type_RNS;
signal   SU                 :  type_RNS;
signal   A1,A2,B1,B2,M1,M2  :  type_RNS;
signal   Gamma              :  type_RNS;
signal   YI1,YI2            :  type_RNS;
signal   ARNS,BRNS,SRNS     :  type_RNS;
signal   Y_ACCU             :  type_YACU;
signal   YI_REG             :  type_YSUM;
-------------------------------------------------------------
component RNSMULT is
Port ( A,B: IN  type_RNS;
        M  : OUT type_RNS
       );
end component RNSMULT;
-------------------------------------------------------------
component ACCU is
Port ( CLK : in std_logic;
        RST :  in std_logic;
        A   :  in type_YSUM;
        ACC :  out type_YACU
       );
end component ACCU;
-------------------------------------------------------------
component CSA is
generic( N : integer := 66);
Port (A1,A2,A3,A4,A5,A6,A7,A8 : in std_logic_vector(N-1 downto 0);
      S                        : out std_logic_vector(N+2 downto 0)
      );
end component CSA;
-------------------------------------------------------------
component BSUM is
Port (A1,A2 : in  type_RNS;
      S      : out type_YSUM );
end component BSUM;
------------------------------------------------------------
component SCMULT is
Port ( A : in  std_logic_vector(w-1    downto 0);
       B : in  std_logic_vector(w-15   downto 0);
       E : out std_logic_vector(2*w-15 downto 0)
       );
end component SCMULT;
-------------------------------------------------------------
component KSUM  is
Port ( A1,A2: in   std_logic_vector(2*w-15 downto 0);
       S    : out  std_logic_vector(2*w-14 downto 0)
       );
end component KSUM;
-------------------------------------------------------------
component RNSSUM is
Port ( A,B: IN  type_RNS;
       S  : OUT type_RNS
     );
end component RNSSUM;
------------------------------------------------------------
begin
----------------------------------------------
-- CUNCURRENT ASSIGNMENTS
----------------------------------------------
U01: RNSMULT  port map(A=> A1, B=>B1, M=>M1);
U02: RNSMULT  port map(A=> A2, B=>B2, M=>M2);
U03: CSA      generic map(8) port map(
     A1=> Gamma(1)(w-1 downto w-8), A2=> Gamma(2)(w-1 downto w-8),
     A3=> Gamma(3)(w-1 downto w-8), A4=> Gamma(4)(w-1 downto w-8),
     A5=> Gamma(5)(w-1 downto w-8), A6=> Gamma(6)(w-1 downto w-8),
     A7=> Gamma(7)(w-1 downto w-8), A8=> Gamma(8)(w-1 downto w-8), S=> AL);
U05: BSUM     port map (A1=> YI1, A2=> YI2, S => YI_REG);
U06: ACCU     port map (CLK => CLKY, RST =>RESETN, A => YI_REG, ACC=> Y_ACCU);
U07: SCMULT   port map (A => G1, B => UD1, E=> K1);
U08: SCMULT   port map (A => G2, B => UD2, E=> K2);
U09: KSUM     port map (A1=> K1_REG, A2=> K2_REG, S=> K_SUM);
U10: RNSSUM   port map (A => ARNS, B=> BRNS, S => SRNS);
Alpha <= ALP(10 downto 8);
-----------------------------------------------------
STATE_MACHINE:
process(CLK,RESETN)
begin
if RESETN='0' then
   STATE  <= T01;
   A1     <= (others=>(others=>'0'));
   A2     <= (others=>(others=>'0'));
   B1     <= (others=>(others=>'0'));
   B2     <= (others=>(others=>'0'));
   AlphaD <= (others=>(others=>'0'));
   SU     <= (others=>(others=>'0'));
   ARNS   <= (others=>(others=>'0'));
   BRNS   <= (others=>(others=>'0'));
   Zi     <= (others=>(others=>'0'));
   ALP    <= (others=>'0');
   G1     <= (others=>'0');
   G2     <= (others=>'0');
   UD1    <= (others=>'0');
   UD2    <= (others=>'0');
   ENY    <= '0';
 --  CLKY    <= '0';
elsif CLK='1' and CLK'event then
  case STATE is
       when T01 =>
            A1 <= Ai;
            B1 <= DINV;
            ENY<='1';
            STATE <= T02;
       when T02 =>
            STATE <= T03;
       when T03 =>
           -- Gamma<= M1;
            STATE <= T04;
       when T04 =>
            A1  <= (others => Gamma(1));
            B1  <= DiMmj(1);
            A2  <= (others => Gamma(2));
            B2  <= DiMmj(2);
            G1  <= Gamma(1);
            G2  <= Gamma(2);
            UD1 <= UDIM(1);
            UD2 <= UDIM(2);
            STATE <= T05;
       when T05 => ------------------------------------------
            STATE <= T06;
       when T06 =>
          --  CLKY <= '1';
            ALP <= AL + DELTA;
            STATE <= T07;
       when T07 =>
            A1  <= (others => Gamma(3));
            B1  <= DiMmj(3);
            A2  <= (others => Gamma(4));
            B2  <= DiMmj(4);
            G1  <= Gamma(3);
            G2  <= Gamma(4);
            UD1 <= UDIM(3);
            UD2 <= UDIM(4);
          --  CLKY <= '0';
            STATE <= T08;
       when T08 =>
            STATE <= T09;
       when T09 =>
            if (Alpha = "000" or Alpha > "111" ) then
                alphaD <= (others=>(others=>'0'));
            else
                alphaD <= Read_MDMmi(Alpha);
            end if;
          --  CLKY  <= '1';
            STATE <= T10;
       when T10 =>
            A1  <= (others => Gamma(5));
            B1  <= DiMmj(5);
            A2  <= (others => Gamma(6));
            B2  <= DiMmj(6);
            G1  <= Gamma(5);
            G2  <= Gamma(6);
            UD1 <= UDIM(5);
            UD2 <= UDIM(6);
          --  CLKY <= '0';
            STATE <= T11;
       when T11 =>
            STATE <= T12;
       when T12 =>
         --   CLKY <= '1';
            STATE <= T13;
       when T13 => -------------------------------------------
            A1  <= (others => Gamma(7));
            B1  <= DiMmj(7);
            A2  <= (others => Gamma(8));
            B2  <= DiMmj(8);
            G1  <= Gamma(7);
            G2  <= Gamma(8);
            UD1 <= UDIM(7);
            UD2 <= UDIM(8);
         --   CLKY <= '0';
            STATE <= T14;
       when T14 =>
            STATE <= T15;
       when T15 =>
         --   CLKY <= '1';
            STATE <= T16;
       when T16 =>
          --  CLKY <= '0';
            STATE <= T17;
       when T17 => --------------------------------------------
            A2 <= (others =>("000000000000000"&K_ACCU(2*w-12 downto w+3)));
            B2 <= NI;   --------- calculate -KM
            STATE <= T18;
       when T18 =>
            YRNS(Y_ACCU,SU);
            STATE <= T19;
       when T19 =>
            ARNS <= AlphaD;
            BRNS <= SU;
            STATE <= T20;
       when T20 =>
            ARNS <= SRNS;
            BRNS <= M2;
            STATE <= T21;
       when T21 => --------------------------------------------
            Zi <= SRNS;
            STATE <= T22;
       when T22 =>
            STATE <= T22;
 end case;
end if;
end process;
-------------------------------------------------
STATE_MACHINE2:
process(CLK,RESETN)
begin
if RESETN='0' or ENY = '0' then
```

```
   ST       <= S01;
   Gamma    <= (others=>(others=>'0'));
  -- Zi       <= (others=>(others=>'0'));
   YI1      <= (others=>(others=>'0'));
   YI2      <= (others=>(others=>'0'));
   KM       <= (others=>(others=>'0'));
   K1_REG    <= (others=>'0');
   K2_REG    <= (others=>'0');
   K_ACCU  <= (others=>'0');
   CLKY <='0';
  -- K        <= (others=>'0');
elsif CLK = '0' and CLK'event then
case ST is
     when S01 =>
          ST <= S02;
     when S02 =>
          ST <= S03;
     when S03 =>
          Gamma <= M1;
          ST <= S04;
     when S04 =>
          ST <= S05;
     when S05 =>
          ST <= S06;
     when S06 =>
          YI1 <= M1;
          YI2 <= M2;
          K1_REG<= K1;
          K2_REG<= K2;
          CLKY <='1';
          ST <= S07;
     when S07 =>
          K_ACCU <= K_ACCU+ K_SUM;
          CLKY <='0';
          ST <= S08;
     when S08 =>
          ST <= S09;
     when S09 =>
          YI1 <= M1;
          YI2 <= M2;
          K1_REG<= K1;
          K2_REG<= K2;
          CLKY <='1';
          ST <= S10;
     when S10 =>
          K_ACCU <= K_ACCU+ K_SUM;
          CLKY <='0';
          ST <= S11;
     when S11 =>
          ST <= S12;
     when S12 =>
          YI1 <= M1;
          YI2 <= M2;
          K1_REG<= K1;
          K2_REG<= K2;
          CLKY <='1';
          ST <= S13;
     when S13 =>
```

```
          K_ACCU <= K_ACCU+ K_SUM;
          CLKY <='0';
          ST <= S14;
     when S14 =>
          ST <= S15;
     when S15 =>
          YI1 <= M1;
          YI2 <= M2;
          K1_REG<= K1;
          K2_REG<= K2;
          CLKY <='1';
          ST <= S16;
     when S16 =>
          K_ACCU <= K_ACCU+ K_SUM;
          CLKY <='0';
          ST <= S17;
     when S17 =>
          ST <= S17;
end case;
end if;
end process;
end Behavioral;
----------------------------------------------------------
-- Written by: Ali Mehrabi
-- Create Date: 02.07.2018 17:14:11
-- Design Name:
-- Module Name: ACCU - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
----------------------------------------------------------
library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.RNS_package.all;

entity ACCU is
Port ( CLK : in std_logic;
       RST : in std_logic;
       A   : in type_YSUM;
       ACC : inout type_YACU
      );
end ACCU;

architecture Behavioral of ACCU is
begin
process(CLK,RST)
begin
if RST = '0' then
   ACC <= (others=>(others=>'0'));
elsif CLK='1'and CLK'event then
  for i in 1 to total_channels loop
     ACC(i) <= ACC(i) + A(i);
  end loop;
end if;
end process;

end Behavioral;

library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.RNS_package.all;


entity BSUM is
Port (A1,A2 : in  type_RNS;
      S     : out type_YSUM );
end BSUM;

architecture Behavioral of BSUM is
begin
MULT: for i in 1 to total_channels generate
      S(i) <= '0'&A1(i) + A2(i);
end generate;

end Behavioral;

library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.RNS_package.all;


entity SCMULT is
Port ( A : in  std_logic_vector(w-1    downto 0);
       B : in  std_logic_vector(w-15   downto 0);
       E : out std_logic_vector(2*w-15 downto 0)
       );
end SCMULT;

architecture Behavioral of SCMULT is

begin
E(2*w-15 downto 0) <= A*B;
end Behavioral;

library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.RNS_package.all;

entity KSUM is
Port ( A1,A2: in   std_logic_vector(2*w-15 downto 0);
       S    : out  std_logic_vector(2*w-14 downto 0)
       );
end KSUM;
```

```
architecture Behavioral of KSUM is

begin
S<= '0'&A1+A2;
end Behavioral;


library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.RNS_package.all;

entity CSA is
generic( N : integer := 66);
Port (A1,A2,A3,A4,A5,A6,A7,A8 : in std_logic_vector(N-1 downto 0);
      S                       : out std_logic_vector(N+2 downto 0)
      );
end CSA;

architecture Behavioral of CSA is

signal P10,P11,P20,P21,G10,G11,G20,G21 :std_logic_vector(N-1 downto 0);
signal S1,S2   :std_logic_vector(N+1 downto 0);
signal C10,C20 :std_logic_vector(N+1 downto 0);
signal C11,C21 :std_logic_vector(N+1 downto 0);
signal S3,S4   :std_logic_vector(N+2 downto 0);

begin
P10 <=  (A1 xor A2);
P11 <=  (A3 xor A4);
G10 <=  (A1 and A2);
G11 <=  (A3 and A4);
S1  <= "00"&(P10 xor P11);
C10 <= '0'&((P10 and P11) or ((not G10) and G11) or (G10 and (not G11)))&'0' ;
C11 <= (G10 and G11) &"00";

P20 <=  (A5 xor A6);
P21 <=  (A7 xor A8);
G20 <=  (A5 and A6);
G21 <=  (A7 and A8);
S2  <= "00"&(P20 xor P21);
C20 <= '0'&((P20 and P21) or ((not G20) and G21) or (G20 and (not G21)))&'0' ;
C21 <= (G20 and G21) &"00";

S3 <= '0'&S1 + C10 + C11;
S4 <= '0'&S2 + C20 + C21;
S <= S3+S4;

end Behavioral;
```

B.5 Test and verification of SOR_2

B.5.1 Python program for generating random test vectors

```
"""
Created on JUN 23 09:56:48 2018
@author: Ali
"""
import numpy as np
import random

p = 2**256-2**32-2**9-2**8-2**7-2**6-2**4-1
m=[]
m.append(2**66-1)
m.append(2**66-2**2-1)
m.append(2**66-2**3-1)
m.append(2**66-2**4-1)
m.append(2**66-2**5-1)
m.append(2**66-2**6-1)
m.append(2**66-2**8-1)
m.append(2**66-2**9-1)

f=open("D:\\test_vectors.txt", 'w')
for j in range (0,10):
    A= random.randint(2**255+1, 2**256-1)
    B =random.randint(2**255+1, 2**256-1)
    C = A*B
    for i in range(7,0,-1):
        d= str('{:066b}'.format((C%m[i])))
        f.write(d)
        f.write('\n')
    for i in range(7,0,-1):
        d =str('{:066b}'.format(((C%p)%m[i])))
        f.write(d)
        f.write('\n')
f.close()
```

B.5.2 Testbench for SOR_2

```
-----------------------------------------
-- Create Date: 23.06.2018 22:48:13
-- Design Name:
-- Module Name: MRTEST_tb - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
-- Revision:
-- Revision 0.01 - File Created

library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.RNS_package.all;
use STD.textio.all;
```

```
entity MRTEST_tb is
end MRTEST_tb;

architecture Behavioral of MRTEST_tb is
component MRTEST is
Port (
CLK: in std_logic;
RESETN: in std_logic;
Ai: in  type_RNS;
Zi: out type_RNS
     );
end component MRTEST;
signal Ai,Zi: type_rns;
signal CLK: std_logic:='0';
signal RESETN: std_logic;
signal aib: std_logic_vector(w-1 downto 0);
file file_VECTORS : text;
begin
ut: MRTEST port map(CLK=>CLK,RESETN=>RESETN,Ai=>Ai,ZI=>ZI);
CLK <= not CLK after 2.67 ns;
process
variable v_ILINE: line;
variable Zi_a, Zi_b: type_rns;
variable Ai_b: std_logic_vector(w-1 downto 0):=(others =>'0');
file file_VECTORS: text open read_mode is "test_vectors.txt";
begin
while not endfile(file_VECTORS) loop
    RESETN<='0';
    wait for 1 ns;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Ai(8) <=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Ai(7) <=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Ai(6) <=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Ai(5) <=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Ai(4) <=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Ai(3) <=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Ai(2) <=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Ai(1) <=Ai_b;
    RESETN<='1';
    wait for 150 ns;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Zi_b(8) :=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Zi_b(7) :=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Zi_b(6) :=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Zi_b(5) :=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Zi_b(4) :=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Zi_b(3) :=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Zi_b(2) :=Ai_b;
    readline(file_VECTORS, v_ILINE);
    read(v_iline, Ai_b);
    Zi_b(1) :=Ai_b;
    Zi_b := Zi;
    if (Zi_a /=Zi_b) then
       report "MISMATCH␣FOUND";
    end if;
end loop;
file_close(file_VECTORS);
wait;
end process;
end Behavioral;
```

B.6 Hardware Description of RNS ECC point multiplication

```
----------------------------------------------------------------------------------
-- Written by: Ali Mehrabi
-- Create Date: 19.07.2018 16:22:27
-- Design Name:
-- Module Name: ECCCORE_ML - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.RNS_PACKAGE.all;
```

```
entity ECC_CORE is
    Port ( Xg     : IN  STD_LOGIC_VECTOR (w-1 downto 0);
           Yg     : IN  STD_LOGIC_VECTOR (w-1 downto 0);
           CLK    : IN  STD_LOGIC;
           CLKIN  : IN  STD_LOGIC;
           CLKOUT : IN  STD_LOGIC;
           RESETN : IN  STD_LOGIC;
           RSTIN  : IN  STD_LOGIC;
           READY  : OUT STD_LOGIC;
           DONE   : OUT  STD_LOGIC;
           INDONE : OUT  STD_LOGIC;
           Xe     : OUT STD_LOGIC_VECTOR (w-1 downto 0);
           Ye     : OUT STD_LOGIC_VECTOR (w-1 downto 0);
           Ze     : OUT STD_LOGIC_VECTOR (w-1 downto 0)
           );
end ECC_CORE;

architecture Behavioral of ECC_CORE is

TYPE STATE_TYPE is (S0,S1,S2,S3,S4,S5,S6,S7,S8);


component ECPA is
Port (CLK   :    in  std_logic;
      RESETN:    in  std_logic;
      X1i,Y1i,Z1i:   in  type_RNS;
      X2i,Y2i,Z2i:   in  type_RNS;
      XOi,YOi,ZOi:   out type_RNS
     );
end component ECPA;

component  ECPD is
Port (CLK:           in  std_logic;
      RESETN:        in  std_logic;
      X1i,Y1i,Z1i:   in  type_RNS;
      X2i,Y2i,Z2i:   out type_RNS
     );
end component ECPD;

component SERIAL_OUT is
Port (CLKOUT: IN  std_logic;
      RST   : IN  std_logic;
      YI    : IN  type_RNS;
      DONE  : OUT std_logic;
      YIS   : OUT std_logic_vector(w-1 downto 0)
      );
end component SERIAL_OUT;

component SERIAL_IN is
Port (CLKIN : IN  std_logic;
      RST   : IN  std_logic;
      DIN   : IN  std_logic_vector(w-1 downto 0);
      DONE  : OUT std_logic;
      YI    : OUT type_RNS
      );
end component SERIAL_IN;
```

```
signal START,CONVERT,DATA_READY,ARESETN,DRESETN :STD_LOGIC;
signal DONE1,DONE2,DONE3 : STD_LOGIC;
signal INDONEX,INDONEY   : STD_LOGIC;
signal SEL : STD_LOGIC_VECTOR(1 downto 0);
signal K   : STD_LOGIC_VECTOR(16*w-1 downto 0);

signal X1,X2,X3,X4,Y1,Y2,Y3,Y4,Z1,Z2,Z3,Z4 : TYPE_RNS;
signal X5,Y5,Z5                            : TYPE_RNS;
signal X1R,Y1R                             : TYPE_RNS;
signal R0X,R0Y,R0Z,R1X,R1Y,R1Z             : TYPE_RNS;
signal STATE: STATE_TYPE;

begin
-------CUNCURRENT ASSIGNMENTS
U1 : ECPA port map(
      CLK =>CLK,
      RESETN=>ARESETN,
      X1i=>X1,Y1i=>Y1,Z1i=>Z1,
      X2i=>X2,Y2i=>Y2,Z2i=>Z2,
      XOi=> X3,YOi=>Y3,ZOi=>Z3
      );
U2 :  ECPD port map(
      CLK => CLK,
      RESETN=>DRESETN,
      X1i=>X4,Y1i=>Y4,Z1i=>Z4,
      X2i=>X5,Y2i=>Y5,Z2i=>Z5
      );
U3:   SERIAL_IN port map(
      CLKIN => CLKIN,
      RST   => RSTIN,
      DIN   => Xg,
      DONE  => INDONEX,
      YI    => X1R
      );
U4:   SERIAL_IN port map(
      CLKIN => CLKIN,
      RST   => RSTIN,
      DIN   => Yg,
      DONE  => INDONEY,
      YI    => Y1R
      );
U5 :  SERIAL_OUT port map(
      CLKOUT=>  CLKOUT,
      RST   =>  RESETN,
      YI    =>  R0X,
      DONE  =>  DONE1,
      YIS   => Xe
      );
U6 :  SERIAL_OUT port map(
      CLKOUT=>  CLKOUT,
      RST =>  RESETN,
      YI  =>  R0Y,
      DONE=>  DONE2,
      YIS => Ye
      );
U7 : SERIAL_OUT port map(
     CLKOUT =>  CLKOUT,
     RST    =>  RESETN,
     YI      =>  R0Z,
     DONE    =>  DONE3,
     YIS     => Ze
     );
DONE <= DONE1 and DONE2 and DONE3;


READ_INPUT_COORDINATES:
process(CLK,RESETN,RSTIN)
begin
if RESETN ='0' or RSTIN='0' then
   INDONE <='0';
   DATA_READY <= '0';
elsif CLK='1' and CLK'event then
   if INDONEX='1' and INDONEY='1' then
      DATA_READY <= '1';
      INDONE <= '1';
   end if;
end if;
end process READ_INPUT_COORDINATES;


CONTROLLER:
process(CLK,RESETN)
variable COUNT :integer;
variable j: integer;
begin
if RESETN = '0' then
   X1  <=(others=>(others=>'0'));
   Y1  <=(others=>(others=>'0'));
   Z1  <=(others=>(others=>'0'));
   X2  <=(others=>(others=>'0'));
   Y2  <=(others=>(others=>'0'));
   Z2  <=(others=>(others=>'0'));
   X4  <=(others=>(others=>'0'));
   Y4  <=(others=>(others=>'0'));
   Z4  <=(others=>(others=>'0'));
   R0X <=(others=>(others=>'0'));
   R0Y <=(others=>(others=>'0'));
   R0Z <=(others=>(others=>'0'));
   R1X <=(others=>(others=>'0'));
   R1Z <=(others=>(others=>'0'));
   ARESETN  <= '0';
   DRESETN  <= '0';
   READY    <= '0';
   STATE    <= S0 ;
   j:=0;
   count:=0;
elsif CLK='0' and CLK'event then
   if DATA_READY = '1' then
      case STATE is
           when S0=>
                j:=MSBB(KEY)-1;---- Private Key(1) is read
                STATE <= S1;
                R0X   <= X1R;   ------P registered in R0
                R0Y   <= Y1R;
                R0Z   <= (others=>"00"&X"0000000000000001");
                X4    <= X1R;   ------ double P
                Y4    <= Y1R;
                Z4    <= (others=>"00"&X"0000000000000001");
           when S1 =>
                DRESETN <='1'; --Do Point Doubling
                STATE   <= S2;
           when S2 =>
                if count <151 then  -- Wait until Point doubling is done
                   count:=count+1;
                   STATE <=S2;
                else
                  count:=0;
                  R1X <= X5;    --2P registered in R1
                  R1Y <= Y5;
                  R1Z <= Z5;
                  STATE <= S3;
                end if;
           when S3 =>
                DRESETN <='0';
                if KEY(j)='1' then --if Ki=1 then R1=2R1, R0=R1+R0 else
                   X4 <= R1X;      --              R0=2R0, R1=R1+R0
                   Y4 <= R1Y;
                   Z4 <= R1Z;
                else
                   X4 <= R0X;
                   Y4 <= R0Y;
                   Z4 <= R0Z;
                end if;
                X1 <= R0X;
                Y1 <= R0Y;
                Z1 <= R0Z;
                X2 <= R1X;
                Y2 <= R1Y;
                Z2 <= R1Z;
                STATE <= S4;
           when S4 =>
                DRESETN <='1';
                ARESETN <='1';
                STATE   <= S5;
           when S5 =>
                if count <240  then
                   count := count +1;
                   STATE  <=S5;
                else
                   count :=0;
                   STATE<=S6;
                end if;
           when S6 =>
                   if KEY(j)='1' then --if Ki=1 then R1=2R1, R0=R1+R0 else
                      R0X <= X3;      --              R0=2R0, R1=R1+R0
                      R0Y <= Y3;
                      R0Z <= Z3;
                      R1X <= X5;
                      R1Y <= Y5;
                      R1Z <= Z5;
                   else
                      R1X <= X3;
                      R1Y <= Y3;
                      R1Z <= Z3;
```

```
                      R0X <= X5;
                      R0Y <= Y5;
                      R0Z <= Z5;
                   end if;
                 STATE<= S7;
           when S7 =>
                ARESETN <='0';       -- Deactivate Point Addition
                DRESETN <='0';       -- Activate point Doubling
                if j>0 then
                   j:= j-1;
                   STATE <= S3;
                 else
                   STATE <= S8;
                 end if;
           when S8  =>
                READY  <='1';
                STATE <=S8;
            end case;
        end if;
    end if;
end process CONTROLLER;
end Behavioral;
```

B.7 Hardware Description of Low-cost RNS GLV ECC point multiplication

```
--------------------------------------------------------
-- Written by : Ali Mehrabi
-- Create Date: 12.10.2019 23:10:56
-- Design Name:
-- Module Name: ECGLV - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 1.01 - File Created
-- Additional Comments:
-- 28/05/2018
--------------------------------------------------------
library IEEE;
library WORK;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use WORK.RNS_PACKAGE.all;
entity  ECCGLV is
   Port (CLK:        in  std_logic;
      RESETN:        in  std_logic;
      A:             in  std_logic_vector(3 downto 0);
      XO:            out std_logic_vector(65 downto 0);
      SEL_O:         in std_logic_vector(4 downto 0);
      DO:            out std_logic;
      AD:            out std_logic;
      DONE:          out std_logic
     );
end  ECCGLV;

architecture Behavioral of ECCGLV is
type   TYPE_STATE is (S0,S1,S2,S3,S4,S5,S6);
constant DPERIOD: integer:= 155;
constant APERIOD: integer:= 370;

signal STATE: TYPE_STATE;
signal  X0,Y0,Z0          : type_RNS;
signal  X1,Y1,Z1          : type_RNS;
signal  X2,Y2,Z2          : type_RNS;
signal  X3,Y3,Z3          : type_RNS;
signal  XR,YR,ZR          : type_RNS;
signal  XS,YS,ZS          : type_RNS;
signal  DRESET,ARESET,SEL  : std_logic;
signal  ADDR, KADDR        : std_logic_vector(3 downto 0);
signal  X,Y,Z             : std_logic_vector(527 downto 0);
signal  K                 : std_logic_vector(515 downto 0);
signal SELD,SELA          : std_logic;
--signal V     : integer; --test point

component SMR is
Port (
      CLK   : IN std_logic;
      RESETN: IN std_logic;
      Ai    : IN type_RNS;
      Zi    : OUT type_RNS
      );
end component SMR ;

component ARITH is
Port (CLK   :     in  std_logic;
      ARESETN:    in  std_logic;
      DRESETN:    in  std_logic;
      XP,YP,ZP:   in  type_RNS;
      X1A,Y1A,Z1A:   in  type_RNS;
      X2A,Y2A,Z2A:   in  type_RNS;
      XOP,YOP,ZOP:   out type_RNS;
      XOA,YOA,ZOA:   out type_RNS
      );
end component ARITH;

component  ECPD is
Port (CLK:           in  std_logic;
      RESETN:        in  std_logic;
      X1i,Y1i,Z1i:   in  type_RNS;
      X2i,Y2i,Z2i:   out type_RNS
     );
end component ECPD;

component ECPA is
Port (CLK   :    in  std_logic;
      RESETN:    in  std_logic;
      X1i,Y1i,Z1i:   in  type_RNS;
      X2i,Y2i,Z2i:   in  type_RNS;
      XOi,YOi,ZOi:   out type_RNS
      );
end component ECPA;

component MUX is
 Port (
 SEL:           in  std_logic;
 X1i,Y1i,Z1i:   in  type_RNS;
 X2i,Y2i,Z2i:   in  type_RNS;
 XOi,YOi,ZOi:   out type_RNS
 );
end component MUX;

component RAMX IS
 port (
   a   : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
   spo : OUT STD_LOGIC_VECTOR(527 DOWNTO 0)
 );
end component RAMX;

component RAMY IS
 port (
   a   : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
   spo : OUT STD_LOGIC_VECTOR(527 DOWNTO 0)
 );
end component RAMY;
component RAMZ IS
 port (
   a   : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
   spo : OUT STD_LOGIC_VECTOR(527 DOWNTO 0)
 );
end component RAMZ;

component RAMK IS
 port (
   a   : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
   spo : OUT STD_LOGIC_VECTOR(515 DOWNTO 0)
 );
end component RAMK;
--- RAM MAPPED AS
--  P   000
--  Q   001
-- -P   010
-- -Q   011
--  P+Q 100
-- -P+Q 101
--  P-Q 110
-- -P-Q 111
begin
XO <= XR(1) when SEL_O = "00000"  else
      XR(2) when SEL_O = "00001"  else
      XR(3) when SEL_O = "00010"  else
      XR(4) when SEL_O = "00011"  else
      XR(5) when SEL_O = "00100"  else
      XR(6) when SEL_O = "00101"  else
      XR(7) when SEL_O = "00110"  else
      XR(8) when SEL_O = "00111"  else
      YR(1) when SEL_O = "01000"  else
      YR(2) when SEL_O = "01001"  else
      YR(3) when SEL_O = "01010"  else
      YR(4) when SEL_O = "01011"  else
      YR(5) when SEL_O = "01100"  else
      YR(6) when SEL_O = "01101"  else
      YR(7) when SEL_O = "01110"  else
      YR(8) when SEL_O = "01111"  else
      ZR(1) when SEL_O = "10000"  else
      ZR(2) when SEL_O = "10001"  else
      ZR(3) when SEL_O = "10010"  else
      ZR(4) when SEL_O = "10011"  else
      ZR(5) when SEL_O = "10100"  else
      ZR(6) when SEL_O = "10101"  else
      ZR(7) when SEL_O = "10110"  else
      ZR(8) when SEL_O = "10111"  else
      (others=>'0');

URAMX: RAMX    port map ( a => ADDR,  spo => X);
URAMY: RAMY    port map ( a => ADDR,  spo => Y);
URAMZ: RAMZ    port map ( a => ADDR,  spo => Z);
URAMK: RAMK    port map ( a => KADDR, spo => K);
UMUX1: MUX     port map ( SEL=> SELD, X1i => XS,Y1i=> YS, Z1i=> ZS,
X2i=> XR, Y2i=>YR, Z2i=>ZR, Xoi=> X1, Yoi=>Y1, Zoi=>Z1);
UMUX2: MUX     port map ( SEL=> SELA, X1i => X2,Y1i=> Y2, Z1i=> Z2,
X2i=> X3, Y2i=>Y3, Z2i=>Z3, Xoi=> XR, Yoi=>YR, Zoi=>ZR);
UAR  : ARITH  port map ( CLK=>CLK ,ARESETN=> ARESET,DRESETN=> DRESET,
XP => X1, YP => Y1, ZP => Z1, X1A => XS, Y1A => YS, Z1A => ZS, X2A => X2,
Y2A => Y2, Z2A =>Z2,
XOP=> X2, YOP => Y2, ZOP => Z2,XOA=> X3, YOA => Y3, ZOA => Z3);
KADDR <= A;
DO    <= DRESET;
AD    <= ARESET;
XS(1) <= X(  w-1 downto 0);
XS(2) <= X(2*w-1 downto w);
XS(3) <= X(3*w-1 downto 2*w);
XS(4) <= X(4*w-1 downto 3*w);
XS(5) <= X(5*w-1 downto 4*w);
XS(6) <= X(6*w-1 downto 5*w);
XS(7) <= X(7*w-1 downto 6*w);
XS(8) <= X(8*w-1 downto 7*w);
YS(1) <= Y(  w-1 downto 0);
YS(2) <= Y(2*w-1 downto w);
YS(3) <= Y(3*w-1 downto 2*w);
YS(4) <= Y(4*w-1 downto 3*w);
YS(5) <= Y(5*w-1 downto 4*w);
YS(6) <= Y(6*w-1 downto 5*w);
YS(7) <= Y(7*w-1 downto 6*w);
YS(8) <= Y(8*w-1 downto 7*w);
ZS(1) <= Z(  w-1 downto 0);
ZS(2) <= Z(2*w-1 downto w);
ZS(3) <= Z(3*w-1 downto 2*w);
ZS(4) <= Z(4*w-1 downto 3*w);
ZS(5) <= Z(5*w-1 downto 4*w);
ZS(6) <= Z(6*w-1 downto 5*w);
ZS(7) <= Z(7*w-1 downto 6*w);
ZS(8) <= Z(8*w-1 downto 7*w);

process(CLK,RESETN)
```

```
variable i     : integer range 3 to 515;
variable count : integer;
begin
  if RESETN = '0' then
     count  := 0;
     STATE  <= S0;
     ARESET <= '0';
     DRESET <= '0';
     SELA   <= '0';
     SELD   <= '0';
     ADDR   <= "0000";
     DONE <= '0';
  elsif CLK='1' and CLK'event then
        case STATE is
            when S0 =>
                i:= 515;
                STATE <= S1;
            when S1 =>
                -- U <= K(i downto i-3);
                 case (K(i downto i-3)) is
                   when "0100" =>       --- select P at ADDR = 000
                        ADDR   <= "0000";
                        STATE  <= S2;
                   when "0001" =>       -- select Q at ADDR 001
                        ADDR   <= "0001";
                        STATE  <= S2;
                   when "0101" =>       -- select P+Q at ADDR 100
                         ADDR  <= "0100";
                         STATE <= S2;
                   when others =>
                         STATE <= S1;
                   end case;
                   if i >=4 then
                      i:= i -4;
                   else
                      STATE <= S6;
                   end if;
            when S2 =>           ----- first doubling
                 SELD  <='0';    ----- ROM OUTPUT to ECPD
                 STATE <= S3;
            when S3 =>
                 DRESET  <= '1'; ------ START DOUBLING
                 if count = DPERIOD  then
                    count := 0;
                    STATE <= S4;
                 else
                   count := count+1;
                 end if;
            when S4 =>
                   SELD   <= '1';
                   DRESET <='0';
                   case K(i downto i-3) is
                          when "0000" =>
                               SELA <= '0';
                               if i>= 4 then
                                  i:=i-4;
                                  STATE <= S3;
                               else
                          STATE <= S6;
                        end if;
                   when "0100"=>
                         SELA <= '1';
                         ADDR <= "0000";
                         STATE <= S5;
                   when "0001"  =>
                         SELA <= '1';
                         ADDR <= "0001";
                         STATE <= S5;
                   when "1000" =>
                         SELA <= '1';
                         ADDR <= "0010";
                         STATE <= S5;
                   when "0010"=>
                         SELA <= '1';
                         ADDR <= "0011";
                         STATE <= S5;
                   when "0101" =>
                         SELA <= '1';
                         ADDR <= "0100";
                         STATE <= S5;
                   when "0110" =>
                         SELA <= '1';
                         ADDR <= "0101";
                         STATE <= S5;
                  when "1001" =>
                         SELA <= '1';
                         ADDR <= "0110";
                         STATE <= S5;
                  when "1010" =>
                         SELA <= '1';
                         ADDR <= "0111";
                         STATE <= S5;
                  when others=>
                         ADDR <= "1111";
           end case;

    when S5 =>
         ARESET <= '1';
         if count = APERIOD  then
            count := 0;
            if i >= 4 then
               i:=i-4;
               SELA <= '1';
               SELD <= '1';
               STATE <= S3;
               ARESET <= '0';
            else
               STATE <= S6;
            end if;
         else
           count := count+1;
         end if;
     when S6 =>
        ARESET <='0';
        DRESET <='0';
        DONE <= '1';
     end case;
end if;
end process;
end Behavioral;
```

References

[1] A. Matthews. *Side-channel attacks on smartcards*. Network Security **2006**(12), 18 (2006). URL http://www.sciencedirect.com/science/article/pii/S1353485806704652.

[2] W. Wunan, C. Hao, and C. Jun. *The attack case of ecdsa on blockchain based on improved simple power analysis*. In X. Sun, Z. Pan, and E. Bertino, eds., *Artificial Intelligence and Security*, pp. 120–132 (Springer International Publishing, Cham, 2019).

[3] S. P. Manuel, S. Victor, and G. Charles. *Side-channel assessment of open source hardware wallets*. Last accessed 10 September 2020, URL https://eprint.iacr.org/2019/401.pdf.

[4] F. Tramer, D. Boneh, and K. G. Paterson. *Remote side-channel attacks on anonymous transactions*. Cryptology ePrint Archive, Report 2020/220 (2020). Last accessed 10 September 2020, URL https://eprint.iacr.org/2020/220.

[5] B. Hettwer, S. Gehrer, and T. Güneysu. *Applications of machine learning techniques in side-channel attacks: a survey*. Journal of Cryptographic Engineering (2019). URL https://doi.org/10.1007/s13389-019-00212-8.

[6] A. Svoboda and M. Valach. *Circuit operators*. Stroje na. Zpracovani Informaci Sb. **111**, 247 (1957).

[7] H. L. Garner. *The residue number system*. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pp. 146–153 (ACM, New York, NY, USA, 1959). URL http://doi.acm.org/10.1145/1457838.1457864.

[8] A. R. Omondi and B. Premkumar. *The Residue Number Systems*. Elliptic curve cryptography (Imperial College press, ISBN-13978-1-86094-866-4, 2007).

[9] P. V. A. Mohan. *Residue Number Systems : Theory and Applications / by P.V. Ananda Mohan* (Cham : Springer International Publishing : Imprint: Birkh auser, 2016).

[10] D. R. Hankerson. *Guide to elliptic curve cryptography / Darrel Hankerson, Scott Vanstone, Alfred J. Menezes*. Elliptic curve cryptography (New York : Springer, New York, 2004).

[11] P. Barrett. *Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor*. In *Proceedings on Advances in cryptology—CRYPTO '86*, pp. 311–323 (Springer-Verlag, London, UK, UK, 1987). URL http://dl.acm.org/citation.cfm?id=36664.36688.

[12] P. L. Montgomery. *Modular multiplication without trial division*. Mathematics of Computation **44** (1985).

[13] V. Bunimov and M. Schimmler. *Area and time efficient modular multiplication of large integers*. Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors. ASAP 2003 pp. 400–409 (2003).

[14] K. C. Posch and R. Posch. *Modulo reduction in residue number systems*. IEEE Trans. Parallel Distrib. Syst. **6**(5), 449 (1995). URL https://doi.org/10.1109/71.382314.

[15] J. Bajard, L. Didier, and P. Kornerup. *An rns montgomery modular multiplication algorithm.* IEEE Transaction on Computers (1998).

[16] P. P. Shenoy and R. Kumaresan. *Fast base extension using a redundant modulus in rns*. IEEE Trans. Comput. **38**(2), 292 (1989). URL https://doi.org/10.1109/12.16508.

[17] J.-C. Bajard, L.-S. Didier, and P. Kornerup. *Modular multiplication and base extensions in residue number systems*. Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001 pp. 59–65 (2001).

[18] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. *Cox-rower architecture for fast parallel montgomery multiplication*. In *Advances in Cryptology-EUROCRYPT 2000. EUROCRYPT 2000. Lecture Notes in Computer Science,*, vol. 1807, pp. 523–538 (2000).

[19] J. Bajard and N. Merkiche. *Double level montgomery cox-rower architecture, new bounds.* In *In Proceedings of the 13th Smart Card Research and Advanced Application Conference*, vol. 47 (2014).

[20] V. S. Miller. *Use of elliptic curves in cryptography*. In H. C. Williams, ed., *Advances in Cryptology — CRYPTO '85 Proceedings*, pp. 417–426 (Springer Berlin Heidelberg, Berlin, Heidelberg, 1986).

[21] N. Koblitz. *Elliptic curve cryptosystems*. Mathematics of Computation **48**(177), 203 (1987).

[22] J. H. Silverman. *The arithmetic of elliptic curves*, vol. 106 (Springer Verlag, 2009).

[23] R. Schoof. *Counting points on elliptic curves over finite fields*. Journal de Theorie des Nombres de Bordeaux **7**(2), 219 (1995). URL http://www.mat.uniroma2.it/~schoof/ctg.pdf.

[24] H. Hasse. *Zur theorie der abstrakten elliptischen funktionenkoerper iii. die struktur des meromorphismenrings. die riemannsche vermutung.* Journal fuer die reine und angewandte Mathematik **175**, 193 (1936). URL http://eudml.org/doc/149968.

[25] B. Smith. *Mapping of elliptic curves*. URL http://www.hyperelliptic.org/tanja/conf/summerschool08/slides/Maps.pdf.

[26] *Explicit formulas database*. URL https://hyperelliptic.org/EFD.

[27] J. M. Pollard. *A monte carlo method for factorization*. BIT Numerical Mathematics **15**(3), 331 (1975). URL https://doi.org/10.1007/BF01933667.

[28] N. Bray. *Lagrange's group theorem*. Last accessed 10 March 2020, URL http://mathworld.wolfram.com/LagrangesGroupTheorem.html.

[29] D. Hankerson and A. Menezes. *Encyclopedia of Cryptography and Security*, pp. 843–844 (Springer US, Boston, MA, 2011). URL https://doi.org/10.1007/978-1-4419-5906-5_255.

[30] *Standards for efficient cryptography sec2: Recommended elliptic curve domain parameters. version 2.0 certicom corp.* (January 27, 2010). Last accessed 10 January 2020, URL https://www.secg.org/sec2-v2.pdf.

[31] M. Lochter and J. Merkle. *Elliptic curve cryptography (ecc) brainpool standard curves and curve generation*. Last accessed 10 March 2020, URL https://tools.ietf.org/pdf/rfc5639.pdf.

[32] M. Mehrabi and C. Doche. *Low-cost, low-power fpga implementation of ed25519 and curve25519 point multiplication*. Information (Switzerland) **10**(9), 1 (2019).

[33] E. Harold. *A normal form for elliptic curves*. In *Bulletin of American Mathematics Society*, vol. 44, pp. 393–422 (2007).

[34] D. J. Bernstein and T. Lange. *Faster addition and doubling on elliptic curves*. In K. Kurosawa, ed., *Advances in Cryptology – ASIACRYPT 2007*, pp. 29–50 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2007).

[35] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. *Twisted edwards curves*. In S. Vaudenay, ed., *Progress in Cryptology – AFRICACRYPT 2008*, pp. 389–405 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008).

[36] P. L. Montgomery. *Speeding the pollard and elliptic curve methods of factorization*. In *MATHEMATICS OF COMPUTATION*, vol. 48, pp. 243–264 (1987).

[37] D. J. Bernstein, T. Lange, and R. Rezaeian Farashahi. *Binary edwards curves*. In E. Oswald and P. Rohatgi, eds., *Cryptographic Hardware and Embedded Systems – CHES 2008*, pp. 244–265 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008).

[38] C. Costello and B. Smith. *Montgomery curves and their arithmetic*. CoRR **abs/1703.01863** (2017). 1703.01863, URL http://arxiv.org/abs/1703.01863.

[39] D. J. Bernstein. *Curve25519: New diffie-hellman speed records*. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, eds., *Public Key Cryptography - PKC 2006*, pp. 207–228 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006).

[40] *Ed25519: High-speed high-security signatures.* Last accessed 30 August 2019, URL https://ed25519.cr.yp.to/.

[41] M. Joye and S.-M. Yen. *The montgomery powering ladder*. In B. S. Kaliski, c. K. Koc, and C. Paar, eds., *Cryptographic Hardware and Embedded Systems - CHES 2002*, pp. 291–302 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2003).

[42] Y. Hao, S. Ma, G. Chen, X. Zhang, H. Chen, and W. Zeng. *Optimization algorithm for scalar multiplication in the elliptic curve cryptography over prime field*. In D.-S. Huang, D. C. Wunsch, D. S. Levine, and K.-H. Jo, eds., *Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues*, pp. 904–911 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008).

[43] M. Huang, K. Gaj, and T. El-Ghazawi. *New hardware architectures for montgomery modular multiplication algorithm*. IEEE Transactions on Computers **60**(7), 923 (2011).

[44] K. Okeya and K. Sakurai. *Fast multi-scalar multiplication methods on elliptic curves with precomputation strategy using montgomery trick*. In B. S. Kaliski, c. K. Koc, and C. Paar, eds., *Cryptographic Hardware and Embedded Systems - CHES 2002*, pp. 564–578 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2003).

[45] A. P. Sawlikar. *Point multiplication methods for elliptic curve cryptography*. International Journal of Engineering and Innovative Technology (IJEIT) (2012).

[46] N. Mukhtar, M. Mehrabi, Y. Kong, and A. Anjum. *Machine-learning-based side-channel evaluation of elliptic-curve cryptographic fpga processor*. Applied Sciences (Switzerland) **9**(1), 1 (2019).

[47] R. P. Gallant, R. J. Lambert, and S. A. Vanstone. *Faster point multiplication on elliptic curves with efficient endomorphisms*. In J. Kilian, ed., *Advances in Cryptology — CRYPTO 2001*, pp. 190–200 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2001).

[48] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition* (Chapman & Hall/CRC, 2012), 2nd ed.

[49] V. S. Dimitrov and T. Cooklev. *Two algorithms for modular exponentiation using non-standard arithmetics*. IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences (1995).

[50] V. Dimitrov, L. Imbert, and P. K. Mishra. *The double-base number system and its application to elliptic curve cryptography*. Mathematics of Computation **77**(262), 1075 (2008). URL http://www.jstor.org/stable/40234547.

[51] C. Doche, D. Kohel, and F. Sica. *Double-base number system for multi-scalar multiplications*. In A. Joux, ed., *Advances in Cryptology - EUROCRYPT 2009 - 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, vol. 5479 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 502–517 (Springer, Springer Nature, United States, 2009).

[52] C. Doche and L. Habsieger. *A tree-based approach for computing double-base chains*. In Y. Mu, W. Susilo, and J. Seberry, eds., *Information security and privacy*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp. 433–446 (Springer, Springer Nature, United States, 2008).

[53] M. Ciet, M. Joye, K. Lauter, and P. L. Montgomery. *Trading inversions for multiplications in elliptic curve cryptography*. Designs, Codes and Cryptography **39**(2), 189 (2006). URL https://doi.org/10.1007/s10623-005-3299-y.

[54] D. Bernstein, C. Chuengsatiansup, and T. Lange. *Double-base scalar multiplication revisited*. Cryptology ePrint Archive (IACR, 2017).

[55] C. Leiva Aburto and N. Theriault. *Optimal 2-3 Chains for Scalar Multiplication*, pp. 89–108 (Springer, 2019).

[56] *7 series fpgas configurable logic block user guide*. Last accessed 10 March 2020, URL https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[57] *7 series fpgas memory resources user guide*. Last accessed 10 March 2020, URL https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[58] R. M. Avanzi. *Side channel attacks on implementations of curve-based cryptographic primitives*. Cryptology ePrint Archive, Report 2005/017 (2005). https://eprint.iacr.org/2005/017.

[59] P. C. Kocher. *Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems*. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pp. 104–113 (Springer-Verlag, London, UK, UK, 1996). URL http://dl.acm.org/citation.cfm?id=646761.706156.

[60] A. Kerckhoffs. *La cryptographie militaire*. URL http://petitcolas.net/kerckhoffs/crypto_militaire_1.pdf.

[61] P. C. Kocher, J. Jaffe, and B. Jun. *Differential power analysis*. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pp. 388–397 (Springer-Verlag, Berlin, Heidelberg, 1999). URL http://dl.acm.org/citation.cfm?id=646764.703989.

[62] M. Bollo and P. Maistri. *Composite fields against side channel analysis for the advanced encryption standard*. IEEE International Conference on, At Marseille (2014).

[63] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. *Power analysis attacks of modular exponentiation in smartcards*. In Ç. K. Koç and C. Paar, eds., *Cryptographic Hardware and Embedded Systems*, pp. 144–157 (Springer Berlin Heidelberg, Berlin, Heidelberg, 1999).

[64] A. Simko. *Models and approaches for differential power analysis*. URL https://www.slideshare.net/Andrejimko/models-and-approaches-for-differential-power-analysis.

[65] S. Chari, J. R. Rao, and P. Rohatgi. *Template attacks*. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 13–28 (Springer, 2002).

[66] J.-L. Danger, S. Guilley, P. Hoogvorst, C. Murdica, and D. Naccache. *A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards*. Journal of Cryptographic Engineering **3**(4), 241 (2013). URL https://hal.inria.fr/hal-00934333.

[67] K. Tiri, M. Akmal, and I. Verbauwhede. *A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smartcards*. Solid-State Circuits Conference. ESSCIRC 2002. pp. 403 – 406 (2002).

[68] A. Shamir. *Protecting smart cards from passive power analysis with detached power supplies*. In *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 71–77 (Springer, 2000).

[69] L. Benini, E. Omerbegovic, A. Macii, M. Poncino, E. Macii, and F. Pro. *Energy-aware design techniques for differential power analysis protection*. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*, pp. 36–41 (IEEE, 2003).

[70] B. Chevallier-Mames, M. Ciet, and M. Joye. *Low-cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity*. IEEE Transactions on Computers **53**(6), 760 (2004).

[71] D. Stebila and N. Thériault. *Unified point addition formulæ and side-channel attacks*. In L. Goubin and M. Matsui, eds., *Cryptographic Hardware and Embedded Systems - CHES 2006*, pp. 354–368 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006).

[72] J. Cheol Ha and S. Jae Moon. *Randomized signed-scalar multiplication of ecc to resist power attacks*. In B. S. Kaliski, ç. K. Koç, and C. Paar, eds., *Cryptographic Hardware and Embedded Systems - CHES 2002*, pp. 551–563 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2003).

[73] E. Oswald and M. J. Aigner. *Randomized addition-subtraction chains as a countermeasure against power attacks*. In *CHES* (2001).

[74] D. May, H. L. Muller, and N. P. Smart. *Random register renaming to foil dpa*. In Ç. K. Koç, D. Naccache, and C. Paar, eds., *Cryptographic Hardware and Embedded Systems — CHES 2001*, pp. 28–38 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2001).

[75] K. Itoh, T. Izu, and M. Takenaka. *Address-bit differential power analysis of cryptographic schemes ok-ecdh and ok-ecdsa*. In *CHES* (2002).

[76] K. Itoh, T. Izu, and M. Takenaka. *A practical countermeasure against address-bit differential power analysis*. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 382–396 (Springer, 2003).

[77] G. Hospodar, B. Gierlichs, E. De Mulder, I. Verbauwhede, and J. Vandewalle. *Machine learning in side-channel analysis: a first study*. Journal of Cryptographic Engineering **1**(4), 293 (2011). URL https://doi.org/10.1007/s13389-011-0023-x.

[78] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning. vol. 1* (2016).

[79] B. Hettwer, S. Gehrer, and T. GÃijneysu. *Deep neural network attribution methods for leakage analysis and symmetric key recovery*. Cryptology ePrint Archive, Report 2019/143 (2019). https://eprint.iacr.org/2019/143.

[80] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (O'Reilly, 2019).

[81] L. Breiman. *Random forests*. Mach. Learn. **45**(1), 5 (2001). URL https://doi.org/10.1023/A:1010933404324.

[82] H. Trevor, T. Robert, and F. Jerome. *The Elements of Statistical Learning* (Springer, 2008).

[83] M. Parsian. *Data Algorithms* (O'Reilly, 2015).

[84] J. Bell. *Machine Learning: Hands-On for Developers and Technical Professionals* (Willy, 2014).

[85] C. M. Bishop. *Neural Networks for Pattern Recognition* (Oxford University Press, Inc., USA, 1995).

[86] K. O'Shea and R. Nash. *An introduction to convolutional neural networks*. CoRR **abs/1511.08458** (2015). 1511.08458, URL http://arxiv.org/abs/1511.08458.

[87] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, and D. J. Inman. *1d convolutional neural networks and applicationsâĂŞa survey*. URL "https://arxiv.org/ftp/arxiv/papers/1905/1905.03554.pdf".

[88] S. Grossberg. *Recurrent neural networks*. Scholarpedia **8**(2), 1888 (2013).

[89] S. Hochreiter and J. Schmidhuber. *Long short-term memory*. Neural Computation **9**(8), 1735 (1997).

[90] H. Maghrebi, T. Portigliatti, and E. Prouff. *Breaking cryptographic implementations using deep learning techniques*. In C. Carlet, M. A. Hasan, and V. Saraswat, eds., *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, vol. 10076 of *Lecture Notes in Computer Science*, pp. 3–26 (Springer, 2016). URL https://doi.org/10.1007/978-3-319-49445-6_1.

[91] J. Masci, U. Meier, D. Cireundefinedan, and J. Schmidhuber. *Stacked convolutional auto-encoders for hierarchical feature extraction*. In *Proceedings of the 21th International Conference on Artificial Neural Networks-Volume Part I*, ICANN'11, pp. 52–59 (Springer-Verlag, Berlin, Heidelberg, 2011).

[92] E. Cagli, C. Dumas, and E. Prouff. *Convolutional Neural Networks with Data Augmentation against Jitter-Based Countermeasures.* In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference* (Taipei, Taiwan, 2017). URL https://hal.archives-ouvertes.fr/hal-01661212.

[93] E. Prouff, R. Strullu, R. Benadjila, E. Cagli, and C. Dumas. *Study of deep learning techniques for side-channel analysis and introduction to ascad database*. Cryptology ePrint Archive, Report 2018/053 (2018). https://eprint.iacr.org/2018/053.

[94] M. Carbone, V. Conin, M.-A. Cornélie, F. Dassance, G. Dufresne, C. Dumas, E. Prouff, and A. Venelli. *Deep learning to evaluate secure rsa implementations*. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**, 132 (2019). URL https://tches.iacr.org/index.php/TCHES/article/view/7388.

[95] B. Timon. *Non-profiled deep learning-based side-channel attacks*. Cryptology ePrint Archive, Report 2018/196 (2018). https://eprint.iacr.org/2018/196.

[96] L. Masure, C. Dumas, and E. Prouff. *A comprehensive study of deep learning for side-channel analysis*. Cryptology ePrint Archive, Report 2019/439 (2019). https://eprint.iacr.org/2019/439.

[97] S. Asif and Y. Kong. *Highly parallel modular multiplier for elliptic curve cryptography in residue number system*. Circuits Syst. Signal Process. **36**(3), 1027 (2017). URL https://doi.org/10.1007/s00034-016-0336-1.

[98] S. Asif. *High-speed low-power Modular arithmetic for elliptic curve cryptosystems based on the residue number system*. Ph.D. thesis, Macquarie university (2016).

[99] J. Bajard, M. Kaihara, and T. Plantard. *Selected rns bases for modular multiplication.* In *In proceedings 19th IEEE Symposium on Computer Arithmetic*, vol. 47, pp. 25–32 (2009).

[100] Z. Cao and X. Wu. *An improvement of the barrett modular reduction algorithm*. International Journal of Computer Mathematics **91**(9), 1874 (2014). https://doi.org/10.1080/00207160.2013.862237, URL https://doi.org/10.1080/00207160.2013.862237.

[101] A. S. Molahosseini, L. S. de Sousa, and C.-H. Chang. *Embedded Systems Design with Special Arithmetic and Number Systems* (Cham : Springer International Publishing : Imprint: Springer, 2017).

[102] A. Karatsuba and Y. Ofman. *Multiplication of many-digital numbers by automatic computers*. Doklady Akad. Nauk SSSR **145**, 293 (1962). Translation in Physics-Doklady 7, 595–596, 1963.

[103] N. Guillermin. *A high speed coprocessor for elliptic curve scalar multiplications over f p*. In *Cryptographic Hardware and Embedded Systems, CHES 2010. CHES 2010. Lecture Notes in Computer Science, vol 6225. Springer,*, vol. 6225, pp. 48–64 (2010).

[104] S. Kawamura, Y. Komano, H. Shimizu, and T. Yonemura. *Rns montgomery reduction algorithms using quadratic residuosity*. Journal of Cryptographic Engineering pp. 1–19 (2018).

[105] G. Paravati, F. Lamberti, F. Gandino, J. Bajard, and P. Montuschi. *An algorithmic and architectural study on montgomery exponentiation in rns*. IEEE Transactions on Computers **61**(08), 1071 (2012).

[106] M. A. Mehrabi. *Improved sum of residues modular multiplication algorithm*. Cryptography **3**(2), 14 (2019). URL https://www.mdpi.com/2410-387X/3/2/14.

[107] B. Phillips, Y. Kong, and Z. Lim. *Highly parallel modular multiplication in the residue number system using sum of residues reduction*. Applicable Algebra in Engineering, Communication and Computing **21**(3), 249 (2010).

[108] S. Asif, M. S. Hossain, Y. Kong, and W. Abdul. *A fully rns based ecc processor*. Integration **61**, 138 (2018).

[109] M. Mehrabi, C. Doche, and A. Jolfaei. *Elliptic curve cryptography point multiplication core for hardware security module*. IEEE Transactions on Computers (2020).

[110] H. Cohen, A. Miyaji, and T. Ono. *Efficient elliptic curve exponentiation using mixed coordinates*. In K. Ohta and D. Pei, eds., *Advances in Cryptology — ASIACRYPT'98*, pp. 51–65 (Springer Berlin Heidelberg, Berlin, Heidelberg, 1998).

[111] D. J. Bernstein and T. Lange. *Faster addition and doubling on elliptic curves*. In K. Kurosawa, ed., *Advances in Cryptology – ASIACRYPT 2007*, pp. 29–50 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2007).

[112] S. Asif, M. Hossain, and Y. Kong. *High-throughput multi-key elliptic curve cryptosystem based on residue number system*. IET Computers and Digital Techniques **11**(5), 165 (2017).

[113] P. M. Matutino, J. Araújo, L. Sousa, and R. Chaves. *Pipelined fpga coprocessor for elliptic curve cryptography based on residue number system*. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 261–268 (2017).

[114] H. Alrimeih and D. Rakhmatov. *Fast and flexible hardware support for ecc over multiple standard prime fields*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **22**(12), 2661 (2014).

[115] M. Esmaeildoust, D. Schinianakis, H. Javashi, T. Stouraitis, and K. Navi. *Efficient rns implementation of elliptic curve point multiplication over gf(p)*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **21**(8), 1545 (2013).

[116] S. T. Schinianakis D. *A rns montgomery multiplication architecture*. IEEE International Symposium of Circuits and Systems (ISCAS) (2011).

[117] J. Lai and C. Huang. *Elixir: High-throughput cost-effective dual-field processors and the design framework for elliptic curve cryptography*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **16**(11), 1567 (2008).

[118] A. P. Fournaris, L. Papachristodoulou, L. Batina, and N. Sklavos. *Residue number system as a side channel and fault injection attack countermeasure in elliptic curve cryptography*. In *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, pp. 1–4 (2016).

[119] R. Selvam and A. Tyagi. *Power side channel resistance of rns secure logic*. In *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, pp. 143–148 (2018).

[120] *Sakura*. URL http://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-X.html.