Systolic Integer Divider for Sunar-Koç ONB Type II multiplier

by

Shubha Muralidhar
B.E., Visveswaraya Technological University, 2011

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Systolic Integer Divider for Sunar-Koç ONB Type II multiplier

by

Shubha Muralidhar
B.E., Visveswaraya Technological University, 2011

Supervisory Committee

Dr. Fayez. Gebali , Supervisor
(Department of Electrical and Computer Engineering)

Dr. Mihai. Sima, Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Fayez. Gebali , Supervisor
(Department of Electrical and Computer Engineering)

Dr. Mihai. Sima, Member
(Department of Electrical and Computer Engineering)

ABSTRACT

This thesis focuses on the Binary Integer Modulo-Division Algorithm that is essential for the permutation process in Sunar-Koç ONB Type II Multiplier and also for other general purposes. This thesis explains the new algorithm developed based on the systolic array architecture which gives a systematic approach to the iterative process for the Modulo-Division. The scheduling and projection timing functions are proposed for the processor array allocation and the matlab code has been implemented to verify the efficiency of the algorithm. The thesis also explores the possibility of word based algorithm for design optimization.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

First and foremost, my sincere gratitude to my supervisor Dr. Fayez Gebali for his constant guidance, encouragement and patience. His vision, expertise and knowledge helped me throughout my MASC degree and also to work on my thesis in a very productive and independent environment. This thesis would not have been possible without his support.

I would also like to thank all my friends and my fellow students working with my supervisor for their unfailing support professionally and personally throughout my studies.

DEDICATION

To my parents, brother and my fiancé

Chapter 1

Introduction

1.1 Finite Field Arithmetic and Background Research

Finite field or Galois field is used in various areas such as computer algebra, number theory, coding theory, digital signal processing and cryptography [1, 2]. This has led to various research in the area for finite field arithmetic operations. The public-key cryptography systems like Elliptic Curve Cryptography use finite field operation such as field multiplication, field exponentiation, field squaring and field inversion. Binary elliptic curve cryptography systems are applicable for smart cards, digital signatures, pseudo-random generators and others. Field multiplication is the most important field operation in the elliptic curve cryptography systems (ECC systems). ECC systems also have the complex field inversion operation which can be carried out by several field multiplication operations. These complex operations along with high number of bits in cryptography makes ECC system implementation very expensive and slow. Hence, it becomes very important to design an efficient low-cost and high-speed finite field multiplier that can significantly improve the performance and the cost of the binary elliptic curve cryptography systems. Over the years, numerous time, area and delay efficient algorithms have been developed by researchers whose performance is based on the finite field basis representation [3–19].

Polynomial/Canonical and Optimal Normal bases are two different forms of finite field basis representation. Whereas, many studies have been done in the area of field multiplication over $GF(2^m)$ with polynomial and binary values, there has been only few studies focussed on multiplication for $GF(p)$ with Integer values. Even fewer

studies have been focussed over $GF(2^m)$ with Optimal Normal bases (ONB). The main advantage of ONB over polynomial bases is that the squaring of the element in ONB is only a right cyclic shift of its binary form. There are two types of Optimal normal bases which are Type I ONB and Type II ONB. Exponentiation of ONB type II is much easier than exponentiation of ONB type I. In recent years, Massey-Omura proposed a multiplication algorithm explained in [4] which works on both Type I ONB and Type II ONB. In [6], an improved and fast architecture for Massey-Omura algorithm was presented which reduced the area and the power consumption for the circuit. Reyhani-Masoleh and Hasan then proposed the sequential normal basis multipliers over $GF(2^m)$ that were AND-efficient and XOR-efficient respectively [7, 8]. These two designs required m clock-cycles to generate the final result. In [9] a multiplexer-based normal basis multiplier algorithm was proposed which used XOR gates and multiplexers instead of AND and XOR in [7] and [8].

Although the hardware design of Type I is more efficient than Type II, it is not preferable in cryptography since the value of m is an even number which raises security concerns. On the other hand, Type II has prime values for m and has been recommended by NIST for cryptography [20]. In 1998, Blake proposed a multiplication method for Optimal normal basis Type II for polynomials of length $2m$ in [10]. This design has an XOR complexity of $(2m)^2$. In 2001, [3]Sunar and Koç proposed an algorithm for ONB Type II which used much lesser hardware compared to Masser-Omura Algorithm and better complexity that required m^2 AND gates and $1.5(m^2-m)$ XOR gates which is 25% lesser XOR gates than Massey-Omura algorithm. Reyhani-Masoleh and Hasan proposed a new architecture based on Massey-Omura ONB parallel multiplier which is applicable to any finite field size [5]. This architecture also reduced the circuit complexity of the Massey-Omura ONB parallel multiplier and gave the same efficiency as that of Sunar-Koç algorithm. In 2015, Systolic array architecture was designed for Sunar-Koç algorithm in [21] which gave a linear and non-linear scheduling of the processor elements and assignment of their tasks. It proposed six systolic array designs. The scheduling and projection techniques proposed in [21] allowed the control of processor workload and the communication flow among the processors.

This led to the motivation of this thesis that is based on the combination of Sunar-Koç algorithm [3] and the systolic array architecture proposed in [21]. Sunar-Koç Algorithm requires Type II ONB be converted into a shifted canonical form based on the range of m before multiplication. The main step to the conversion is obtaining

an Integer modulo of the index values of the binary form of finite element over the range of m. There has never been any systematic approach in finding this modulo in a fast way. Therefore, it becomes very important to design a high speed Integer modulo-divider which can carry out this conversion that aids in Sunar-Koç ONB type II multiplier. As part of this thesis, such a high speed Integer divider algorithm has been designed and a systolic array architecture has been proposed based on the iterative procedure of the algorithm.

1.2 Contributions

This thesis introduces Binary Integer Modulo-Division algorithm which is required for basis conversion in Sunar and Koç ONB Type II Multiplier. Following are its contributions:

1. Propose Systolic architecture for expressing binary integer modulo-division algorithm in an iterative fashion.
2. Linear Scheduling and Projection function applied to map the iteration to processor array.
3. Algorithm verification using Matlab.
4. Proposed n-linear directed acyclic graph to optimise the design.

1.3 Organisation of the Thesis

The organisation of thesis is as follows:

In **Chapter 2**, we present the mathematical background about Finite Field. We explain the laws of finite field, different basis in finite field, Sunar-Koç ONB Type II Multiplier Algorithm over $GF(2^m)$ and its permutation process.

In **Chapter 3**, we introduce the Bit-wise Binary Integer Modulo-Division algorithm and explain the mathematics behind the algorithm. We also explain the iterative process and introduce a systolic architecture for the algorithm.

In **Chapter 4** we explore the scheduling and projection function for the design and mapping to processors.

In **Chapter 5** we show the hardware design for the permutation process using integer modulo-division algorithm. We also include the matlab implementation of the algorithm and the simulation result in it.

In **Chapter 6** we discuss the optmization of the design through word based approach for the design and explain the scheduling function based on it.

In **Chapter 7** we conclude the thesis and show the future research and implementation path.

Chapter 2

Mathematical Background

In this chapter we explain the mathematics behind the finite fields and finite field arithmetic. We, also explain the different types of finite fields and their representation in the mathematical format. We then understand the Sunar and Koç ONB type II multiplier algorithm and it's process.

2.1 Finite Field

Finite Field in mathematics also known as Galois field (GF) is a field which contains finite number of elements and on which the addition, subtraction, multiplication and division operations are defined and satisfy the basic rules similar to any rational numbers and real numbers. The field is defined as a set of $\mathbb{F}$ with two operations, addition and multiplication which are denoted by + and . , respectively, that satisfy the following laws:

1. For all a and b in $\mathbb{F}$, $a + b$ and $a.b$ are all in $\mathbb{F}$.
2. $(a + b) + c = a + (b + c)$ and $(a.b).c = a.(b.c)$ (Associative law for addition and multiplication).
3. $a + b = b + a$ and $a.b = b.a$ (Commutative law for addition and multiplication).
4. There exists an additive identity element 0 in $\mathbb{F}$ such that $a + 0 = a$ for all a.
5. There exists a multiplicative identity element 1 in $\mathbb{F}$ such that $a.1 = a$ for all a.
6. There is an element $-a$ in $\mathbb{F}$ such that $a + (-a) = 0$.

7. There is an element a^{-1} in $\mathbb{F}$ such that $a.a^{-1} = 1$ for any a$\neq$0.

8. For all a,b and c in $\mathbb{F}$, $a.(b+c) = (a.b)+(a.c)$ (Distributive law of multiplication over addition).

The Galois field is also denoted as $GF(p)$ where p is either a prime number or power of prime number. The number of elements in the field is called its order. Therefore, the order of $GF(p)$ is p and it can have its elements in the range $\{0, 1, ..., p-1\}$. The finite field $GF(2)$ thus consists of elements 0 and 1. So it can be simply said that p can be constructed with the elements that are modulo of p. This modulo operation is required to satisfy all the rules of the Galois field. For example, the addition and multiplication operation on $GF(2)$ is given by:

Table 2.1: $GF(2)$ Modulo Operation

a	b	addition(+)	multiplication(.)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The extension field $GF(p^m)$ can be constructed over this using a polynomial modular arithmetic over $GF(p)$. Every element in this extension field is represented by a polynomial of degree $m-1$. The modulo operation then becomes the polynomial modulo operation an irreducible degree of order m over $GF(p)$. Therefore, there always exists a basis of $\{\alpha_0, \alpha_1, ..., \alpha_{m-1}\}$ in a way that an element a in $GF(p^m)$ is represented by:

$$a = a_0\alpha_0, a_1\alpha_1, ..., a_{m-1}\alpha_{m-1} \tag{2.1}$$

where, $a_i \in GF(p)$

2.2 Types of Basis in Finite field

2.2.1 Polynomial or Canonical Basis

For a $GF(2^m)$ which is an m-dimensional vector space over $GF(2)$, we need m different linearly independent elements for the elements of $GF(2^m)$. This is the basis of the

vector space. The most commonly used basis of the field arithmetic operations in the Galois field $GF(2^m)$ with the ordered set $\{1, \beta, \beta^2, ..., \beta^{m-1}\}$ where $\beta \in GF(2^m)$ is the polynomial basis or also known as canonical basis.

2.2.2 Optimal Normal Basis

Optimal Normal Basis is known for being very efficient for hardware implementation in $GF(2^m)$ because of low power consumption compared to other bases. It has various other advantages including in the field of cryptography.

The ordered set of ONB can be represented by $\{\beta, \beta^2, \beta^4, ..., \beta^{2^{m-1}}\}$ where $\beta \in GF(2^m)$ forms the normal basis of the field arithmetic. β here is called as a normal element.

There are two types of Optimal normal bases namely, optimal normal basis of type I and the optimal normal basis of type II as classified in [22]. Type II has an advantage over type I ONB because the value of m is preferred to be a prime number for security concerns during cryptography. The type II ONB having this feature of prime values for m has been recommended by NIST for cryptography [20] [23].

The optimal normal basis of Type II of the $GF(2^m)$ is formed by the normal element $\beta = \mu + \mu^{-1}$ where μ is a primitive $(2m+1)^{th}$ root of unity which mean $\mu^{2m+1} = 1$ and $\mu^i \neq 1$ for all $1 \leq i < 2m+1$.

ONB Type II is formed by m and p where $p = 2m + 1$ is a prime and also either of the two conditions below are true:

• 2 is a primitive root modulo p. This means every integer value that is a co-prime to p is congruent to power 2 modulo p given by equation (2.2) as:

$$2^k = a(\mathrm{mod} p) \tag{2.2}$$

where, k is an integer.

• $p = 7(\mathrm{mod} 8)$ and the multiplicative order of 2 modulo p is m which is given by equation (2.3) as:

$$2^m = 1(\mathrm{mod} p) \tag{2.3}$$

The basis can be represented as:

$$M = \{\mu + \mu^{-1}, \mu^2 + \mu^{-2}, \mu^{2^2} + \mu^{-2^2}, ..., \mu^{2^{m-1}} + \mu^{-2^{m-1}}\} \tag{2.4}$$

which can also be given as:

$$M = \{\beta, \beta^2, \beta^{2^2}, ..., \beta^{2^{m-1}}\} \tag{2.5}$$

2.3 Sunar-Koç ONB Type II Multiplier

In 2001, Sunar and Koç proposed an ONB Type II multiplier based on the fact that there is a normal element $\beta = \mu + \mu^{-1}$ where μ is a primitive $(2m+1)^{th}$ root of unity [3]. This multiplier designed by Sunar and koç had better speed and needed less hardware compared to other designs.The computation for Sunar-Koç requires only m^2 AND gates and $1.5(m^2 - m)$ XOR gates which is 25% fewer XOR gates than the well-known Massey-Omura multiplier algorithm [4].

Consider two elements A and B that need to be multiplied and are given in the ONB Type II basis, M as:

$$M = \{\beta, \beta^2, \beta^4, ..., \beta^{2^{m-1}}\} \tag{2.6}$$

In order to multiply these two elements using Sunar-Koç algorithm, both the elements need to be converted into the shifted canonical basis N which is obtained by a permutation of the basis elements in M and then the two elements are multiplied in the new basis N.

The shifted canonical basis is of the form:

$$N = \{\mu + \mu^{-1}, \mu^2 + \mu^{-2}, \mu^3 + \mu^{-3}, ..., \mu^m + \mu^{-m}\} \tag{2.7}$$

which can also be given as:

$$N = \{\beta, \beta^2, \beta^3, ..., \beta^m\} \tag{2.8}$$

The two cases that satisfy ONB conditions are examined below:

1. **If 2 is a primitive root modulo p ($2^k = a(\mathrm{mod} p)$):**

   Let us consider P_1 a set of powers of 2 modulo p in the range [1,2m] :

$$P_1 = \{2, 2^2, 2^3, ..., 2^{2m-1}, 2^{2m}\} \pmod{p} \tag{2.9}$$

   After modulo operation over set P_1 by p, equation (2.9) becomes:

$$Q_1 = \{1, 2, 3, 4, ...2m\} \tag{2.10}$$

Therefore, it shows that the basis element which was in the form of $\mu^{2^i} + \mu^{-2^i}$ can be written as $\mu^j + \mu^{-j}$. Also, if $j \geq m+1$ then it is always possible to write $\mu^j + \mu^{-j}$ as $\mu^{(2m+1)-j} + \mu^{-(2m+1)+j}$ which brings the power of μ to the range $[1, m]$.

2. **If the multiplicative order of 2 modulo p is m ($2^m = 1(\mathrm{mod} p)$):**

   The set P_2 of powers of 2 modulo p can be written as:

$$P_2 = \{2, 2^2, 2^3, ..., 2^{m-1}, 2^m\} \pmod{p} \tag{2.11}$$

   Set P_2 now consists of m distinct integers that are in the range $[1, 2m]$. In order to bring these integers in the range $[1, m]$, we need to follow the two conditions below:

   (a) $1 < 2^i \bmod p \leq m$: In this case, $2^i \bmod p$ is already in the range [1, m]. So we leave it as it is.

   (b) $m < 2^i \bmod p < 2m$: In this case, the number $(2m + 1) - (2^i \bmod p)$ is written in its place to bring it to the range [1, m].

   The set P_2 now becomes equivalent to:

$$Q_2 = \{1, 2, 3, 4, ...m\} \tag{2.12}$$

   As a result of this, the basis element $\mu^{2^i} + \mu^{-2^i}$ for $i \in [1, m]$ can be written as $\mu^j + \mu^{-j}$ for $j \in [1, m]$.

Through the above process, the basis M which is in the form in equation (2.6) can now yield a set which is in shifted canonical basis N form.

The above process is the Permutation process in Sunar-Koç algorithm.

Let us consider an element A in the finite field that can be represented in the basis M as:

$$A = a_1^{'} v + a_2^{'} v^2 + a_3^{'} v^{2^2} + ... + a_m^{'} v^{2^{m-1}} \tag{2.13}$$

where, $v^{(i)} = (\mu + \mu^{-1})^{(i)}$ for $1 \leq i < 2^{m-1}$ and, $a_i^{'} \in GF(2^m)$

Its equivalent permutation element in basis N is given by:

$$A = a_1 v_1 + a_2 v_2 + a_3 v_3 + ... + a_m v_m \tag{2.14}$$

where, $v_{(i)} = (\mu + \mu^{-1})^{(i)}$ for $1 \leq i < m$

The permutation between coefficients a_j and $a_i{}'$ can be expressed as:

$$j = \begin{cases} k \quad \text{if} \quad k \in [1, m], \\ (2m+1) - k \quad \text{if} \quad k \in [m+1, 2m], \end{cases} \tag{2.15}$$

where, $k = 2^{i-1} \mod (2m+1)$ for $i = 1, 2, \cdots, m$. Permutation process is the most crucial part of the Sunar-Koç ONB Type II multiplier. This permutation process converts the elements that need to be multiplied from the optimal normal basis to a shifted canonical basis. The same permutation can be also used for inverse permutation to convert the basis back to M from basis N.

The systolic array architecture in [21] uses these elements which are in shifted canonical form as operands for the ONB multiplier.

The multiplication algorithm designed by Sunar and koç can thus be seen as consisting of three main steps:

1. Convert operands A and B represented in optimal normal basis M to shifted canonical basis N using permutation.

2. Multiply operands A and B in the basis N. The systolic array architecture in [21] can be used for this.

3. Convert the result back to basis M using inverse permutation.

2.3.1 Example for Permutation Process

This section illustrates how the permutation process is used to convert between ONB type II basis M and shifted canonical basis N for the field $GF(2^5)$. Since $m = 5$, we have $p = 2m + 1 = 11$ and 2 is primitive in modulo 11. There also exists M which is an optimal normal basis type II for the field $GF(2^5)$ that can be represented as $M = \beta, \beta^2, \beta^4, \beta^8, \beta^{16}$, where $\beta = \mu + \mu^{-1}$. The exponents of M are {1,2,4,8,16} and we use the identity $\mu^{11} = 1$ to convert the basis from M to N. Exponents 1, 2 and 4 are in the range [1,5], so we leave it as it is. Exponent 16 can be brought to the range by taking a modulo over p as $16 \mod 11 = 5$. It is now in the range [1,5]. We now

need to bring exponent 8 to the range using the identity $\mu^8 = \mu^{8-11} = \mu^{-3}$. Therefore we get,

$$\beta = \mu + \mu^{-1} = \mu + \mu^{-1} = \beta_1, \tag{2.16}$$

$$\beta^2 = \mu^2 + \mu^{-2} = \mu^2 + \mu^{-2} = \beta_2, \tag{2.17}$$

$$\beta^4 = \mu^4 + \mu^{-4} = \mu^4 + \mu^{-4} = \beta_4, \tag{2.18}$$

$$\beta^8 = \mu^8 + \mu^{-8} = \mu^{-3} + \mu^3 = \beta_3, \tag{2.19}$$

$$\beta^{16} = \mu^{16} + \mu^{-16} = \mu^5 + \mu^{-5} = \beta_5, \tag{2.20}$$

The new converted basis in the form N is now $\{\beta_1, \beta_2, \beta_4, \beta_3, \beta_5\}$. It is clear from the example above and from the two conditions of ONB, that the permutation process requires modulo p operation to bring the integers of the elements in the range $[1, m]$ to convert the basis between M and N. Modulo operation is the remainder obtained when one element is divided by another. In this case, we need to divide the m distinct integer values by the divisor p. This motivates us to develop a high speed Integer modulo-divider which yields good speed and low cost.The next chapter will discuss all about this Integer modulo-divider algorithm using a Systolic Array architecture for fast binary integer division.

2.4 Chapter Summary

This chapter explained the mathematical background of Finite field, different types of Finite field basis and also gave a brief description of the Sunar-Koç Permutation algorithm. It further discusses the Modulo Division Algorithm using systolic architecture which is explained in the next chapter.

Chapter 3

Modulo-Division algorithm

The Permutation process in Sunar-koç multiplier requires to bring elements of the set into the range $[1, m]$. This is done by taking a modulo of 2 raised to the powers of index values of the elements in the set over p where $p = 2m+1$. The modulo operation is nothing but a division with a non-zero remainder. The index values of the elements are made up of integer values and therefore, an Integer divider was developed which can find the modulo over p. This chapter explains how the Integer Modulo-Division algorithm is designed which is not only applicable in Sunar-Koç permutation process but also for general purpose Integer division.

3.1 Integer Division

There have been various binary division algorithms. The basic idea behind binary hardware divider is to arrive at the correct quotient and remainder through an iterative process. Assume we have two positive integers; an n-bit length dividend A and an m-bit length divisor B. The division equation would then be given by:

$$A = qB + r \tag{3.1}$$

where, q and r are quotient and remainder, respectively.

The quotient can also be expressed as:

$$q = \left\lfloor \frac{A}{B} \right\rfloor \tag{3.2}$$

The remainder r is an integer in the range $0 \leq r < B$ and is given as:

$$r = A - qB \tag{3.3}$$

The Integer division algorithm can be executed in two phases. **Phase (1)** includes iterative process based on the m and n values similar to the basic long division and updates the value of Y and Z as follows:

$$Y \leftarrow Y - \delta X \tag{3.4}$$
$$Z \leftarrow Z + \delta \tag{3.5}$$

where, δ is the partial quotient which is subtracted from Y and added to Z.

The number of iterations in **Phase (1)** is given by $n - m + 1$. Thus **Phase (1)** of the division algorithm can be shown as:

$$Y^{(i+1)} = Y^{(i)} - \mu_i \delta^{(i)} B \tag{3.6}$$
$$Z^{(i+1)} = Z^{(i)} + \mu_i \delta^{(i)} \tag{3.7}$$

where $Y^{(i)}$ represents the intermediate remainder at iteration i, $Z^{(i)}$ represents the intermediate quotient at iteration i and the initialization condition is:

$$Y^{(0)} = A \tag{3.8}$$
$$Z^{(0)} = 0 \tag{3.9}$$

The iteration variables μ_i and $\delta^{(i)}$ are specified by:

$$\mu_i = \begin{cases} 1 & \text{when} \quad Y^{(i)} \geq 0 \\ -1 & \text{when} \quad Y^{(i)} < 0 \end{cases} \tag{3.10}$$
$$\delta^{(i)} = 2^{(n-1-m-i)}, \quad 0 \leq i < n - m \tag{3.11}$$

Phase (2) is the post processing of Y and Z values in order to bring the value of Y in the range $0 \leq Y^{(n-m+1)} < B$ since **Phase (1)** may yield a result of Y which is either negative or positive and is still greater than B. The value for Z should also be calculated whenever Y is calculated.

In this thesis, we approach another condition in phase other than addition or

subtraction of the δB which is called a no-operation. For no-operation condition to satisfy the xor operation of μ and the first bit of the updated value of A should yield 0. That is given by equation:

$$nop = \begin{cases} 1 & \text{when} \quad \mu_k \oplus A^{(k)}(1) = 0 \\ 0 & \text{when} \quad \mu_k \oplus A^{(k)}(1) = 1 \end{cases} \tag{3.12}$$

When (3.12) evaluates to 1, we need to perform logical left shift operation on A by 1 position and that gives the updated value for A. This reduces the computation time which otherwise requires to add/subtract δB from A. For every updated shift value of A, the new calculated value for C would be the subtraction of δ from previous value of C.

Algorithm 1 Bit-wise Integer Modulo-Division algorithm. μ is the sign of A.

1: Input: A, B, δ;
2: $Y \leftarrow A$; $Z \leftarrow 0$; $\delta = \delta$; %initialize
3: $\mu = sign(A)$;
4: **for** $k = 0 : m - n$ **do**
5: $nop = \mu^{(k)} \oplus A^{(k)}(1)$;
6: **if** $nop \neq 0$ **then**
7: $A = A << 1$;
8: $shift = 1$
9: **else**
10: $Y^{(k+1)} \leftarrow Y^{(k)} - \mu^{(k)}\ \delta^{(k)}\ B$;
11: **end if**
12: $Z^{(k+1)} \leftarrow Z^{(k)} + \mu^{(k)}\ \delta^{k}\ shift$;
13: **end for**
14: Output: Y, Z;

Algorithm 1 specifies the Integer Modulo-Division algorithm in which line 5 determines the no-operation based on the MSB value and the sign of the partial results of Y. Lines 9 and line 10 calculate the partial quotient and remainder values for quotient, Z and remainder, Y respectively.

3.2 Systolic Array Architecture of Integer Modulo-Division Algorithm

The fastest way to produce any arithmetic result is through parallel computing where multiple data elements are fed to the hardware architecture, the data is processed and

the result is generated in parallel fashion. Multiple processors operate simultaneously through alternative communication and computation process to provide parallelism in processing multiple data.

Systolic Array Architecture is one such parallel computer architecture where array of processors form a tight mesh like network called cells or nodes. Each node is capable of performing some operations, generating partial result and store that result. The data fed to the processors are computed and this data flows among adjacent processors in one or more directions. Systolic array architecture provides a simple communication between cells in a one, two or three-dimensional structure architecture [24] and the data needs to be provided or received only at the boundary cells [25]. This is one of the main advantage of systolic array architecture because of which external caches or buses are not required to store intermediate results. Thus, this kind of parallel architecture reduces the area, cost and speed to a great extent compared to other architecture. No previous work has been focussed on generating such a systematic processor architecture for a hardware Integer division algorithm.

The data flow in the Systolic array architecture between the processors gives rise to the data dependency because the data input at iteration i is the data output from iteration $i-1$. The data is fed into the systolic array architecture through data bus. At each iteration in the systolic array the data is stored in D-flip flops which is clock synthesised making sure the timing is satisfied and the data is supplied to the nodes at the same time. The data dependency graph for such a systolic array architecture is shown in Figure 3.1.

Let us consider A and B as two integers which are 4-bits and 15-bits each including the sign bit. Figure 3.1 shows the data dependence graph for algorithm (1) where A is the dividend, B is the divisor, q_i is the partial quotient and r_i is the remainder. The maximum length along i-axis is the number of iteration and is given by $i = m - n + 1$ and the maximum length along j-axis is given by the length of a which is n-bits.

Each circle in the figure indicates a cell/node. These nodes are connected in a two-dimensional mesh topology. The communication between the nodes are shown by a straight lines that connects the nodes. The inputs are fed at the boundary cells on the top and on the right. With each iteration, partial result has been generated which is passed on to the nodes below and the result from the left boundary cell indicated in red circles generate partial quotient. The red circles also determine the no-operation which is the line 4 of algorithm (1). The blue circles indicate the partial division which is the line 9 and line 11 of algorithm (1). After i iteration the final Q and R

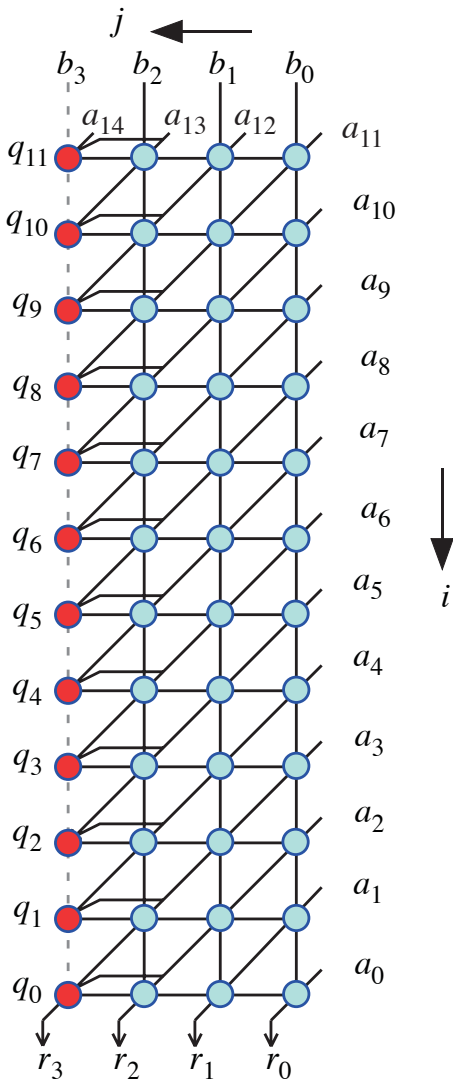

Figure 3.1: Data dependency graph for Integer Modulo-Division algorithm for $m = 15$, $n = 4$ and $i = 12$.

are obtained from the bottom boundary cells of the dependence graph simultaneously.

The main bottleneck for such an array architecture is the problem on System input/output bandwidth in parallel supply of large amount of input data since the data bus is not always as wide as the data that needs to be fed into the systolic array. This can be solved by having a broadcast bus at the input which in this case on top and the right side of the array architecture that feeds to all the input boundary nodes at a time. Arrays of memory storage with large bandwidth can be used along with the broadcast bus to send the data in parallel into the input nodes.

The same issue is found even at the output on System input/output bandwidth in receiving large amount of parallel data. Again same strategy to have the broadcast bus with arrays of memory storage with large bandwidth can be implemented to avoid this bottleneck issue.

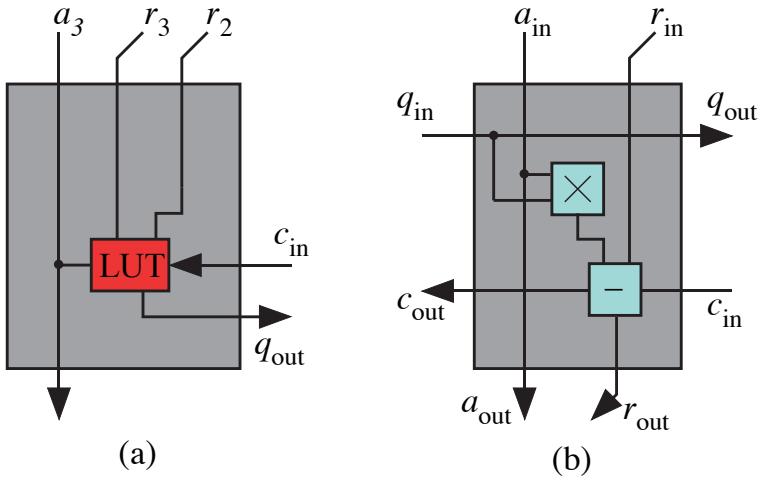

Figure 3.2: Cell diagram showing details of red and blue circles.

For every i^{th} iteration, the q_{out} in Figure 3.2(a) is calculated based on the Look-up Table (LUT) in Table 3.1 which takes three inputs, one sign-bit from the divisor b and first two bits from the partial remainder, r from $i-1$ iteration and produces the partial quotient given out as:

The q_{out} from the red circle is used by the blue circles as shown in Figure 3.2(b) which takes it as an input and multiplies it with input a_{in}. The result is subtracted from the partial remainder, r_{in} obtained from $i-1$ iteration. The blue circle thus indicates a modular multiply/accumulate (MAC) operations.

Each red cell generates one bit of q_{out} corresponding to line 11 of algorithm (1).

Table 3.1: LUT for calculation of q_{out} based on values of a and r for $n = 4$.

b_3	r_2	r_1	q_{out}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

It generates the bit from the most significant digit.

3.3 Chapter Summary

This chapter explained the basic Integer division algorithm, dependence graph and the bit-wise scheduling for the algorithm. Note that for every ith iteration, the calculation cannot proceed without the partial remainder, r from $i-1$ iteration. This gives rise to some of the limitations in the algorithm. It is important to understand the scheduling of each iteration in the algorithm. The next chapter explains in detail the design space exploration for Modulo-Division algorithm

Chapter 4

Design Space Exploration Using Scheduling and Projection

This sections explores the scheduling and the projection of the design for Integer Modulo-Division algorithm based on the dependency graph of Figure 3.1. The approach to the timing function is based on the theory given in [24].

4.1 Scheduling Function

For a given point $p(i, j)$ in the dependency graph in Figure 3.1, the time, t associated with that point is given by the scheduling function $\vec{s} = \begin{bmatrix} s_1 & s_2 \end{bmatrix}$ where:

$$t(p) = sp = is_1 + js_2 \tag{4.1}$$

There are two limitations over the choices for $\vec{s}$ based on the data flow in Figure 3.1.

1. The calculation at point $p_1(i, j)$ must be executed before the calculation at point $p_2(1 + 1, j)$ where, $0 \leq j < n - 1$ and this can be written as:

$$t(p_2) = t(p_1) + 1 \tag{4.2}$$

$$\begin{bmatrix} s_1 & s_2 \end{bmatrix} \begin{bmatrix} i+1 \\ n-1 \end{bmatrix} = \begin{bmatrix} s_1 & s_2 \end{bmatrix} \begin{bmatrix} i \\ n-1 \end{bmatrix} + 1 \tag{4.3}$$

$$s_1(i+1) + s_2(n-1) = s_1 i + s_2(n-1) + 1 \tag{4.4}$$

$$s_1 = 1 \tag{4.5}$$

The scheduling vector thus becomes $\vec{s} = \begin{bmatrix} 1 & s_2 \end{bmatrix}$

2. The calculation at point $p_1(i,j)$ must be executed at the same time as calculation at point $p_2(i,j+1)$ where, $0 \leq j < n-1$ and this can be written as:

$$\begin{bmatrix} 1 & s_2 \end{bmatrix} \begin{bmatrix} i \\ n-1 \end{bmatrix} = \begin{bmatrix} 1 & s_2 \end{bmatrix} \begin{bmatrix} i \\ n-2 \end{bmatrix} \tag{4.6}$$

$$i + s_2(n-1) = i + s_2(n-2) \tag{4.7}$$

$$s_2[n-1-n+2] = 0 \tag{4.8}$$

$$s_2 = 0 \tag{4.9}$$

The scheduling vector thus becomes $\vec{s} = \begin{bmatrix} 1 & 0 \end{bmatrix}$

The result of the scheduling vector $\vec{s} = \begin{bmatrix} s_1 & s_2 \end{bmatrix}$ is given by Figure 4.1

The scheduling vector enables the simultaneous calculation of the partial remainders. It produces the final modulo output R simultaneously after $m-j$ iterations.

4.2 Projection Function

This sections shows how each point in the dependency graph in Figure 3.1 can have time index values associated with them as shown in Figure 4.1. Each node in the Figure 4.1 can be assigned to a processor in the array space. The projection of a point p in the Figure 4.1 maps to a processor in the processor array space. The projection direction for this can then be given by:

$$\vec{d} = \begin{bmatrix} 1 & 0 \end{bmatrix}^t \tag{4.10}$$

If we assume two point in the dependency graph lie along the same projection direction $\vec{d}$ then these two points will be mapped to the same processor. With the projection

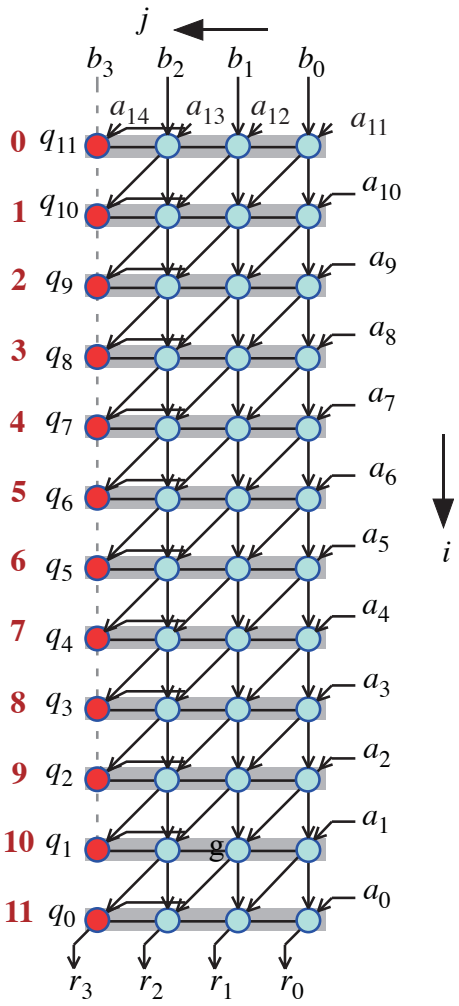

Figure 4.1: Timing function for Integer Modulo-Division algorithm with $m = 15$ and $n = 4$ derived from dependency graph in Figure 3.1.

direction being the null-vector of the projection matrix P can be given as:

$$P = \begin{bmatrix} 0 & 1 \end{bmatrix} \tag{4.11}$$

where P is a rank-deficient projection matrix based on the projection direction $\vec{d}$. If we seek to have an one-dimensional processor array to implement the Integer Modulo-Divider algorithm then P will reduce to a row vector. A projected point $\overline{p}$ in the processor array space corresponding to the point p in the dependency graph space becomes:

$$\overline{p} = Pp = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = j \tag{4.12}$$

Equation (4.12) shows that each scalar value of j in Figure 4.1 corresponds to a processor index in the processor array that it maps to.

The only restriction that need to be ensured is that $sd \neq 0$ where s is a scheduling vector and d is a projection direction vector as given in [24] which requires that the work load is divided among the processor in the processor array per clock cycle and all the processors are utilised at each time step.

The scheduling vector s and the projection direction vector d for Integer Modulo-Divider algorithm is given in Table 4.1.

Table 4.1: Scheduling and projection direction vector for Integer Modulo-Divider algorithm

Scheduling Vector, $\vec{s}$	Projection Direction Vector, $\vec{d}$
$\vec{s} = \begin{bmatrix} 1 & 0 \end{bmatrix}$	$\vec{d} = \begin{bmatrix} 1 & 0 \end{bmatrix}^t$

4.3 Processing Element(PE)

Processing elements form the building block of the design. Array of such processing elements is what makes up the processor array space. From equation (4.12) we see that for the Integer Modulo-Divider algorithm with $m = 15$ and $n = 4$, all the nodes in a column map to a single PE and each column will execute at a different time step based on our scheduling function given by $\vec{s} = \begin{bmatrix} 1 & 0 \end{bmatrix}$. Thus the number of

processing elements is given by j from the equation (4.12). Since, $j = 4$ when $n = 4$, at a given time t there are four processor elements that make up the processor array space. This is shown in the Figure 4.2(a).

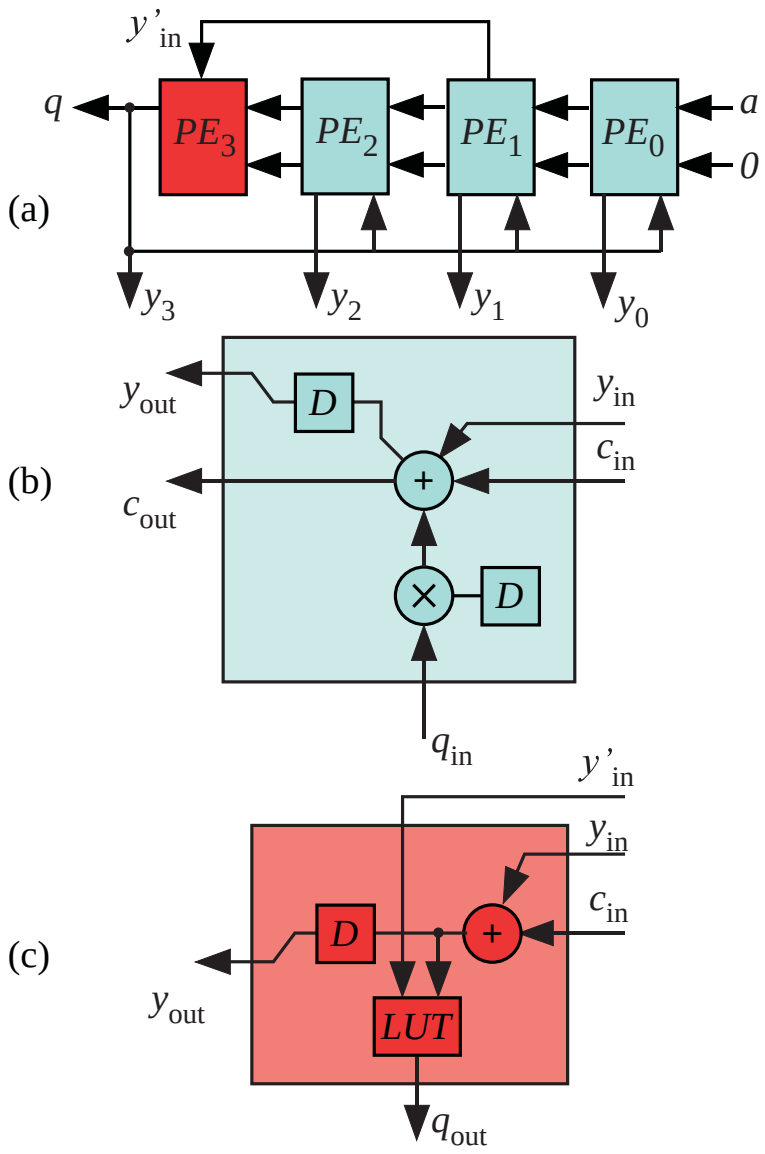

Figure 4.2: Hardware details for Processor array activity.

Figure 4.2(b) shows the processing element for blue circle in the Figure 4.2(a). This processing element is made up of Full adder, AND gates and Flip-flops which can be used to implement the multiplier and the adder required to implement the Integer Modulo-Divider algorithm and this is as shown in Figure 4.2(b). The D flip-

flops hold the data that needs to be used for the next iteration. These flip-flops also hold the data for b_{in} which needs to be multiplied with the q generated from the red circles in the Figure 4.2(a) and added to a_{in} using the full adder to obtain the quotient, q_{out} and the remainder, r_{out} for the next time iteration. The full adder is a 3-bit adder that takes a_{in}, c_{in} and the product of b_{in} and q_{in} as inputs and gives out sum and carry-out. The sum is the partial remainder and the carry-out is passed to the

The red Processing element in the Figure 4.2(a) generates the q_{in} from the XOR gate that implements the 2-LUT based on the look-up table from Table 3.1 and is shown in Figure 4.2(c). This takes two inputs y_{in} and $y_{in}^{'}$. The carry-in that is generated is ignored in this stage.

4.4 Chapter Summary

In this chapter, the timing functions such as scheduling and projection of the finite element is explained which maps the iterative sections of the Figure 4.1 into the processor array for the hardware implementation which is explained in the next section.

Chapter 5

Design Overview, Implementation and Simulation

This chapter explains how the Sunar-koç permutation has been implemented using Modulo-Division algorithm explained in the Chapter 3.

5.1 Hardware design for Permutation Algorithm

Figure 5.1 shows the block diagram of the Permutation algorithm given by Sunar and Koç [3]. The index value of each bit is used as an input to the left shift register(LSR) which is basically a multiplier which converts the index value into powers of 2. The Modulo-Divider takes this input as a Dividend and the value of p where, $p = 2m + 1$ as a divisor and generates the modulo output. If the value obtained is in the range $[1, m]$, we keep the index as it is. If it is not in the range then the new index value is obtained by subtracting the Modulo-Divider output from p.

5.2 Design Implementation using Matlab

This section explains the software implementation of the Modulo-Division algorithm described in the Section (3) of this Thesis using Matlab. Various tests and calculations were done to verify the design thoroughly to ensure there are no bugs in the design and that the numerical result obtained using this Matlab code is accurate. The Full Matlab code can be found in Appendix A.1.

In this code two inputs, divisor $x0 = 127773$ and the dividend $y0 = 338579150$ are

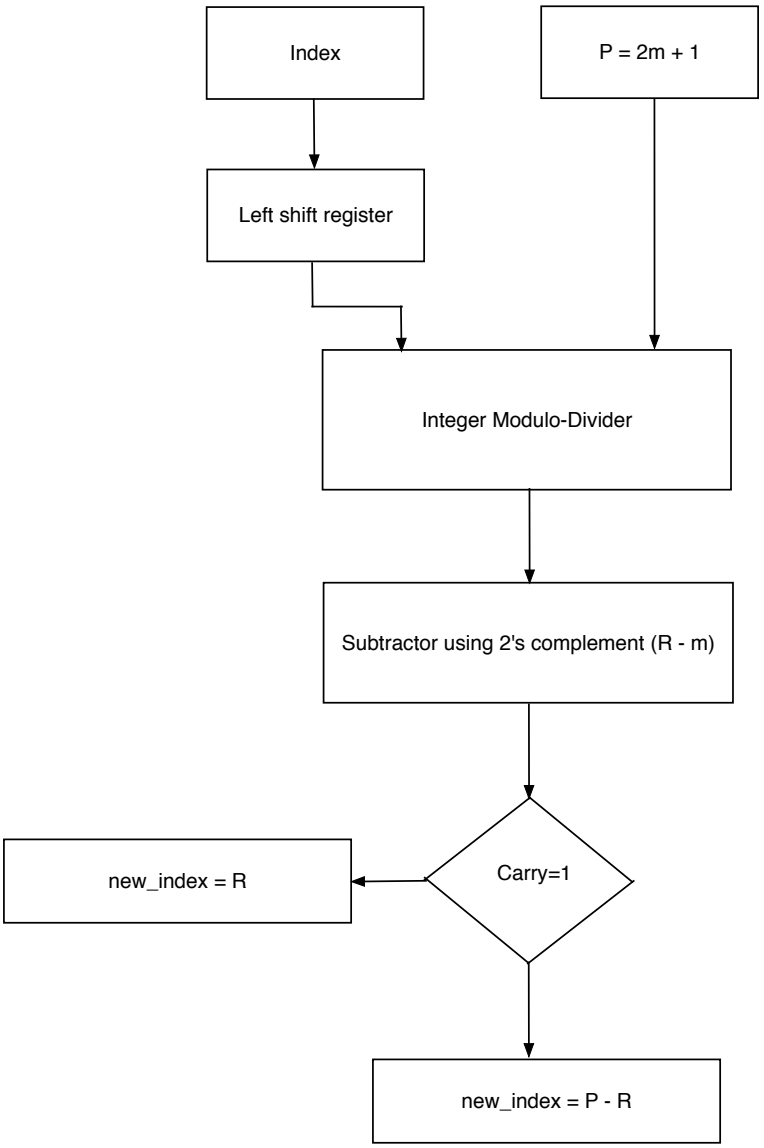

Figure 5.1: Permutation Hardware design using Integer Modulo-Divider.

used as an example to verify the generation of accurate modulo and division result. Two main functions are implemented. The first function is func_print_2complement which converts $x0$ and $y0$ into binary numbers using a matlab function `dec2bin`. The code snippet for this function is shown below:

```
function x_bin = func_print_2complement(x,n)
% print a number x in 2's complement format
% using n bits


if x >= 0
    if x > 2^n-1
        fprintf('func_print_2complement:')
        fprintf('x = %d is bigger than 2^%d-1 = %d\n',x, n-1,
            2^(n-1)-1);
        return;
    end
    x_bin = dec2bin(x,n);
else
    if x < -2^(n-1)
        fprintf('func_print_2complement:')
        fprintf('x = %d is smaller than -2^%d = %d\n',x, n,
           -2^(n-1));
        return;
    end
    x = 2^n + x;
    x_bin = dec2bin(x+2^n,n);
end
```

This function also makes sure that the size of the number to be converted into binary is within the range and gives out error if this condition is not satisfied.

The second function that is instantiated in the main matlab code is the func_divide_algorithm. Once the binary value of $x0$ and $y0$ are obtained the *delta* is calculated based on the size of the two inputs to the Modulo-Divider which is given by $delta = n - m - i$. func_divide_algorithm takes inputs x, y, z and *delta* and generates y_out, z_out and $shift\_out$. This func_divide_algorithm is instantiated $i = n - m$

times with initial value of z as 0. The code snippet for func_divide_algorithm is shown below:

```
function [y_out,z_out, mu, mu_delta,shift_out] = ...
    func_divide_algorithm1(x,y_in,z_in,delta,i,shift)

mu = sign(y_in);

y_bin = func_print_2complement(y_in,32-i);
%y_bin = sscanf(dec2bin(y_in),'%1d');

%y_bin_shift = sscanf(dec2bin(bitshift(a,1)),'%1d')

if (xor(mu,bin2dec(y_bin(1)))) == 1
    fprintf('No operation\n');
    y_bin_shift = func_print_2complement((bitshift(y_in,1))
       ,32-i+1);
    y_out_shift = y_bin_shift(1:end-1)
    y_out = bin2dec(y_out_shift);
   shift_out = 1;
else
    y_out = y_in - mu*x*delta;
    y_out_bin = dec2bin(y_out)
    shift_out = 0;
end

%y_out = y_in - mu*x*delta;
if shift == 1
    z_out = z_in - mu*delta;
else
    fprintf('No shift\n');
    z_out = z_in + mu*delta;
end


mu_delta = mu*delta;
```

This function is instantiated i times for i iterations. With every iteration, the output of the function which has generated partial remainder and the partial quotient is fed back into the function to generate the final output after i iterations.

The function func_divide_algorithm extracts the sign of x using a built-in-function **sign** and assigns it to μ. This value is xor'd with the MSB of y to check for shift operation using a built-in **xor** function. Based on this XOR value the output y_{out} is generated either shifted one bit left using the matlab syntax $y\_bin\_shift(1 : end - 1)$ or it is calculated as $y\_out = y\_in - \mu * x * \delta$. The output z_out is generated by either subtracting/adding $\mu * \delta$ from/to z_in based on shift operation required or not respectively.

After $n - m$ iteration of divide algorithm function, the post processing of Y and Z is done where Y and Z are the partial remainder and the quotient respectively. In the post processing x_0 is added to Y if $Y < 0$ and subtracted from Y if $Y > x_0$. Respectively, Z is incremented or decremented by 1.

The above code was to implement the Integer Modulo-Division Algorithm. In order to verify it, the built-in **mod** function and **division** function in matlab are used. The values generated are rounded to their integer values using built-in matlab **round** function. The code snippet this modulo-division using built-in matlab function is shown below:

```
y_calc  = mod (y0,x0); % remainder
z_calc = floor(y0/x0); % quotient
fprintf('
    ------------------------------------------------------------------\n');
fprintf ('True calucated values:\n');
fprintf('y_calc\t\t = %d\t\t'   , y_calc);
fprintf('z_calc\t\t = %d\n\n', z_calc);

y_calc_int = round(y_calc*2^(n-2));
% y_calc is a fraction. Need to scale is up.
z_calc_int = round(z_calc*2^(n-2));
```

The values obtained from the built-in matlab division and mod functions are compared to the value generated from the algorithm code. Both the values matched perfectly thus ensuring the correctness of the algorithm designed.

5.3 Numerical Simulation using Matlab

To illustrate the operation of the Modulo-Division algorithm, the numerical simulation was done with various values. One main advantage of this algorithm is the shift operation. Shift operation helps reduce the number of calculations in each iteration. If the XOR operation of the sign of the partial quotient and it's MSB yields result as 0, the iteration has no calculation rather the partial remainder is just shifted left by one bit. The corresponding partial quotient is calculated.

Table 5.1 shows the result of the algorithm without the shift operation and Table 5.2 shows the result of the algorithm with the shift operation for the values for X and Y as:

$$X = 127773 \quad and \quad Y = 338579150 \tag{5.1}$$

The result obtained is shown in the below tables:

Table 5.1: Matlab Output of Integer Modulo-Divider with no-shift operation for $m = 32$ and $n = 17$.

i	Y	Z	δ	μ
0	338579150	0	16384	1
1	-1754853682	16384	8192	-1
2	-708137266	8192	4096	-1
3	-184779058	4096	2048	-1
4	76900046	2048	1024	1
5	-53939506	3072	512	-1
6	11480270	2560	256	1
7	-21229618	2816	128	-1
8	-4874674	2688	64	-1
9	3302798	2624	32	1
10	-785938	2656	16	-1
11	1258430	2640	8	1
12	236246	2648	4	1
13	-274846	2652	2	-1
14	-19300	2650	1	-1
Output of Step 1	108473	2649		
Output of Step 2	108473	2649	0	0

Table 5.2: Matlab Output of Integer Modulo-Divider with shift operation for $m = 32$ and $n = 17$.

i	Y	Z	δ	μ
0	338579150	0	8192	1
3	76900046	2048	1024	1
5	11480270	2560	256	1
8	3302798	2624	32	1
10	1258430	2640	8	1
11	236246	2648	4	1
Output of Step 1	108473	2649		
Output of Step 2	108473	2649	0	0

$$R = 338579150 \quad mod \quad 127773 = 108473 \tag{5.2}$$

$$Q = 338579150 \quad div \quad 127773 = 2649 \tag{5.3}$$

Table(5.1) shows that the algorithm without the shift operation takes 14 iterations of arithmetic operations inducing more delay and cost whereas the algorithm with the shift operation shown in Table(5.2) takes only 6 iterations for arithmetic operations using multiplier and adder and rest of the iterations are implemented using a simple left shift operation making it a very fast divider.

5.4 Chapter Summary

In this chapter, the hardware design for the permutation algorithm was shown and the Matlab code was developed to verify the algorithm. The result was illustrated to prove the correct functionality and fast approach to obtained the mod and division operation of two integer numbers. For any design that has been developed, the correct optimization factors, become a very important factor. The next section explains the optimization of the modulo-division algorithm.

Chapter 6

Optimization

This chapter explores the new design optimization method. The algorithm and the dependency graph explained in Chapter 3 are a bit-wise algorithms where every iteration reduces the input by one bit. This limitation can be exploited to further explore the design to have word-based dependency graph.

6.1 Scheduling of word-based Modulo Division Algorithm

The word-based modulo division algorithm requires the w-bit of data input A and B to be passed into the systolic array at every time-step instead of passing one-bit at a time and it produces a result in a word-serial order. This results in performance of a w iterations at the same time as opposed to 1 iteration at a time.

Figure 6.1 shows the dependency graph derived from the bit-wise dependence graph and word size assumed as $w = 3$. This word size is not fixed and is the choice of the designer while implementing such a word-based modulo-divider. The number of iterations is given by $floor((m - n + 1), w)$ which is shown as a row indices which is also a time index values. For any given point $p(i, j)$ in the dependence graph in Figure 6.1, the scheduling function can be represented as:

$$t(p) = \left\lfloor \frac{i}{w} \right\rfloor \tag{6.1}$$

where, $0 \leq i \leq m - n + 1$.

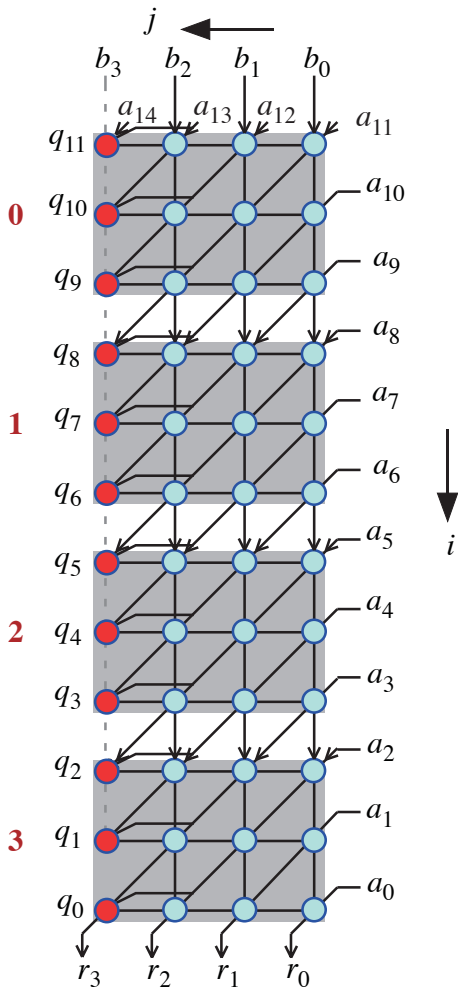

Figure 6.1: Directed acyclic graph or a word-based Integer Modulo-Division algorithm for $m = 15$, $n = 4$ and $w = 3$ derived from dependence graph in Figure 3.1.

Figure 6.1 shows the case where, $m-n$ is an integer multiple of w. For the cases where $m-n$ is not an integer multiple of w, the input to the algorithm is appended with 0s. This adds additional rows to the dependence graph and makes it an integer multiple of w. It is always ensured that the outputs are updated and extracted one word at a time.

6.2 Hardware implementation of word-based algorithm

This section shows how the hardware can be implemented with the optimised approach of word-based/non-linear Integer Modulo-Division algorithm. From equation (6.1) it can be seen that for each given time there is always $\lfloor \frac{i}{w} \rfloor$ bits of inputs fed into the divider. The scheduling function and the projection function are same for this kind of design. For a given point, p in Figure 6.1 is given by:

$$p = \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i & j \end{bmatrix}^t \tag{6.2}$$

The projection matrix P is a two-dimensional matrix and can be given by:

$$P = \begin{bmatrix} \bmod w & 0 \\ 0 & 1 \end{bmatrix} \tag{6.3}$$

A projected point $\overline{p}$ in the processor array space corresponding to the point p in the dependence graph space becomes:

$$\overline{p} = Pp = \begin{bmatrix} \bmod w & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i \bmod w \\ j \end{bmatrix} \tag{6.4}$$

The projection direction vector $\vec{d}$ for the non-linear based algorithm is now different from that of the linear algorithm. It will now not collapse into a single row instead it is now a two-dimensional matrix having ($i \mod w$) rows and j columns.

This equation shows that for a given time, w bits of input data is processed together to generate the quotient and the remainder. The $\lfloor \frac{i}{w} \rfloor$-bit input for divisor and dividend is processed through a framework of full adders and multipliers which generates $\lfloor \frac{i}{w} \rfloor$-bits quotient and $\lfloor \frac{i}{w} \rfloor$-bits remainder. Figure 6.2 shows hardware implementation for such a word-based algorithm with with $m = 15$, $n = 4$ and $w = 3$.

The D flip-flops are used to store the result from the previous iteration. The framework is made up of 2-bit and 3-bit full adders, 2-bit multiplier, D flip-flops and a LUT. The 2-bit full adder and LUT form the red part of the figure where the quotient is calculated in the red-circled full adder. This quotient is then fed to the blue circled multpliers. These blue circled 2-bit multipliers multiply the quotient value and the divisor b and sends the product to the blue circled adders that generates sum and carry. The sum generated at this time-step is the partial remainder. The carry generated is propogated through the adder and is ignored at the red circled adder. The red coloured LUTs are used for shift operation which does XOR operation on the value from the sign bit of dividend a and MSB of a. If the XOR value is '0' then the dividend a value is shifted left by one bit and the addition and multiplication in that iteration is skipped reducing the time required to generate the final result.ss

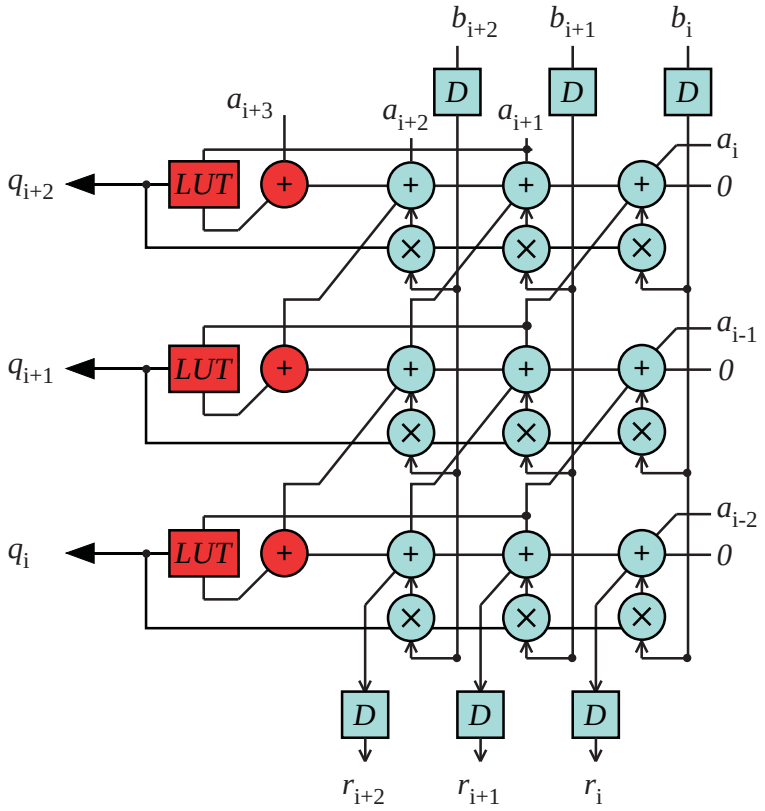

Figure 6.2: Hardware details for a word-based Integer Modulo-Division algorithm with $m = 15$, $n = 4$ and $w = 3$.

6.3 Chapter summary

This chapter explored another approach to design the modulo division algorithm using a word based design which could optimize the time. It also explored the scheduling and projection function for such a design and explained how the hardware could be implemented for it. This is a proposal towards yielding a better performance of the design.

Chapter 7

Conclusions

This thesis presented an Integer Modulo Division Algorithm which can be used for various applications including implementation of the Permutation process in Sunar and kocç algorithm to convert the basis from Optimal Normal Basis type II to shifted canonical basis. The Division algorithm was verified by implementing the Matlab Code and running the simulation with real data which resulted in accurate result making this algorithm very efficient. The Numerical simulations were provided which supported the fact that the shift operation suggested in the division algorithm avoided computation in each iteration step thus reducing the cost of such an algorithm.

Adhoc techniques develop hardware for the division operation directly from the algorithm designed. In this thesis, with reference to [24], a systematic technique was adapted in designed the hardware for the Integer Modulo-Division algorithm designed. The step-wise approach includes:

1. Convert the algorithm into iterations
2. Obtain data dependency graph for systolic array architecture. This step is very different from the normal theory since this had a systematic approach. Each variable was investigated separately which has never been done by anyone before for hardware division algorithm.
3. Design scheduling function for the systolic array architecture
4. Design projection function for the systolic array architecture

Such a design is scalable for different sizes of the divisor and dividend of any Integer divider provided it is scaled at the design process for the required size and

cannot be scaled during a run time. This makes it very suitable in applications like cryptography where the word-length for $GF(2^m)$ varies. The main bottleneck for Systolic Array architecture on the system Input/output bandwidth was discussed and essential solution was provided. The optimisation technique for a word-based Integer modulo-divider was suggested which could implement a faster divider with bigger architecture. Further work could be based on this optimisation proposed for the word-based Integer Modulo Divider.

Appendix A

Matlab Code

A.1 Main Division Code

The below mentioned code is the main matlab code which takes two input $x0 = 127773$ and $y0 = 338579150$ that are of size $m = 18$-bits and $n = 32$-bits respectively and divides them to give out the final quotient and modulo result. It calls two functions, func_divide_algorithm and func_print_2complement. func_divide_algorithm code is given in appendix(A.2). It is a divider function called iteratively i times where $i = n - m + 1$. Each iteration produces partial remainder and quotient values. The second function, func_print_2complement code given in appendix(A.3) is used to convert values of $x0$ and $y0$ into their binary forms.

```
clear all
clc
close all

% program to divide two integers using Algorithm 1:
% the fixed delta based on the iterations.

n = 32;
m = 18;
hex_places_n = n/4;
hex_places_m = ceil(m/4);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%x = 127773; %q
%x = 27773;
% Ensure that 0 < y0 < 2^n-1
%y0 = abs(round(floor(rand))) * (2^n-3)+1;
%y0 = x + (2^(n-m+5));
%y0 = 1176349; % this will only need phase 1
%y0 = 106790;
x0 =  127773; % q value is 17 bits and highest power is 2^16
%y0 = 2072086837; % seed 0
y0 = 338579150;  % seed 1
%y0 = 1749629467;  % seed 2
%y0 = 1945185409;  % seed 3
if (y0 >=2^(2*n)-2)
    fprintf('main_divide_fixed: y0 is out of bound ')
    fprintf('2^n = %d ', 2^n);
    fprintf('y0 = %d ', y0);
    fprintf('2^n = %d ', 2^n);
end

x0_hex = dec2hex(x0,n/4);
y0_hex = dec2hex(y0,n/4);

fprintf('Inputs are:\n')
fprintf('x0\t = %d\t'   , x0);
fprintf('(HEX): %s\n',x0_hex);
fprintf('y0\t = %d\t'   , y0);
fprintf('(HEX): %s\n',y0_hex);

x0_bin = func_print_2complement(x0,m)
y0_bin = func_print_2complement(y0,n)

% initialize arrays y and z. Due to having to do
% n-m+1 iterations, we have one extra value for y and z
% to store the output of each iteration.
y          = zeros(1,n-m+1);
```

```
z           = zeros(1,n-m+1);
shift       = zeros(1,n-m+1);
delta       = ones(1,n-m);
delta_exp   = zeros(1,n-m);
mu          = zeros(1,n-m);

% Initialize array delta so that highest
% value of left shift is n-m = 14 due to:
% 1. Most significant one in x=q is at location
%    17 corresponding to 2^(16).
% 2. Data size is 32 with bit at location 32
%    is the sign bit -2^(31).
% 3. Maximum value of left shift is 14 locations
%    to move MSB of x to location 31.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Highest power of Y could be 2^30 since M = 2^31-1.
% delta really should be between 2^0 to 2^14
% which comes to exponent between 0 to n-m-1
for i = 1:n-m % n-m = 32-17-1 = 14
    delta_exp(i) = n-m-i; % 14, 13, ..., 0
    delta(i) = 2^(delta_exp(i));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Content of register at start of each iteration:
% Register contents at iteration 0:
y(1) = y0;
z(1) = 0;
shift(1) = 0;

for i = 1:n-m % we need n-m itertions
    [y(i+1),z(i+1), mu(i), mu_delta,shift(i+1)] = ...
        func_divide_algorithm1(x0,y(i),z(i),delta(i),i,shift(
            i));
    % Note that func_divide_algorithm1 makes decisions
```

```
83      % based on values of y value stored in the register
84      % i.e. value of y upon entering the iteration.
85      % This value is stored in Y_reg.
86
87  %%shift(i) = shift(i+1);
88
89  if y(i) < 0
90          temp = 2^n + y(i);
91  else
92          temp = y(i);
93  end
94  y_in = dec2hex(temp,8);
95  if y(i+1) < 0
96          temp = 2^n + y(i+1);
97  else
98          temp = y(i+1);
99  end
100 y_out = dec2hex(temp,8);
101
102 if z(i) < 0
103         temp = 2^n + z(i);
104 else
105         temp = z(i);
106 end
107 z_in = dec2hex(temp,8);
108 if z(i+1) < 0
109         temp = 2^n + z(i+1);
110 else
111         temp = z(i+1);
112 end
113 z_out = dec2hex(temp,8);
114
115 counter = n-m-i;
116
117 fprintf('iteration %d\t'        , i-1);
```

```
118 fprintf('counter %d\n'          , counter);
119 fprintf('y_in\t   = %10d (hex:%s)\t\t', y(i),y_in);
120 fprintf('y_out\t  = %10d (hex:%s)\n'    , y(i+1),y_out);
121 fprintf('z_in\t   = %10d (hex:%s)\t\t', z(i),z_in);
122 fprintf('z_out\t  = %10d (hex: %s)\n' , z(i+1),z_out);
123 fprintf('delta_exp  = %10d\t', delta_exp(i));
124 fprintf('mu*delta     = %10d\t' , mu(i)*delta(i));
125 fprintf('mu*delta*x0 = %10d\n\n', mu(i)*delta(i)*x0);
126
127 end
128
129 y_step2 = y(n-m+1);
130 z_step2 = z(n-m+1);
131 delta_step2 = 0;
132 mu_step2 = 0;
133
134
135 % check to see if we need step 2:
136
137 % Note that we make decision about going to Step 2
138 % based on y value produced by the ALU (not stored in
      register yet.
139 % i.e. value of y upon leaving the iteration.
140 % This value is not stored in Y_reg.
141 fprintf('-------------------------------------------------------\n')
142 fprintf('Entering Step 2 of Algorithm 2\n')
143 if ((y(n-m+1) > 0) && (y(n-m+1)< x0))
144     y_step2 = y(n-m+1);
145     z_step2 = z(n-m+1);
146     mu_step2 = 0;
147     delta_step2 = 0;
148     fprintf('y (%d)> 0 AND y(%d) < x0\n',(n-m+1), (n-m+1));
149     fprintf('y_in = %d\ty_out = %d\n',y(n-m+1),y_step2);
150     fprintf('z_in = %d\tz_out = %d\n',z(i),z_step2);
151 elseif y(n-m+1) < 0
```

```
152         y_step2 = y(n-m+1) + x0;
153         z_step2 = z(n-m+1) -1;
154         mu_step2 = -1;
155         delta_step2 = 1;
156         fprintf('y (%d)< 0, we try to increase it\n',(n-m+1));
157         fprintf('y_in = %d\ty_out = %d\n',y(n-m+1),y_step2);
158         fprintf('z_in = %d\tz_out = %d\n',z(n-m+1),z_step2);
159         pause
160     elseif y(n-m+1) >= x0
161         y_step2 = y(n-m+1) -x0;
162         z_step2 = z(n-m+1) + 1;
163         mu_step2 = 1;
164         delta_step2 = 1;
165         fprintf('y (%d) >= x0, we try to decrease it\n',(n-m+1));
166         fprintf('y_in = %d\ty_out = %d\n',y(n-m+1),y_step2);
167         fprintf('z_in = %d\tz_out = %d\n',z(n-m+1),z_step2);
168     end
169
170
171     % Calculate and print the correct results in decimal and Hex
172     y_calc  = mod (y0,x0); % remainder
173     z_calc = floor(y0/x0); % quotient
174     fprintf('
        ------------------------------------------------------------\n');
175     fprintf ('True calucated values:\n');
176     fprintf('y_calc\t\t = %d\t\t'   , y_calc);
177     fprintf('z_calc\t\t = %d\n\n', z_calc);
178
179     y_calc_int = round(y_calc*2^(n-2));
180     % y_calc is a fraction. Need to scale is up.
181     z_calc_int = round(z_calc*2^(n-2));
182
183     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
184     % Now print the iterations in hex for debugging
185     fprintf('------------------------------------------------\n\n')
```

```
186 fprintf('i\t');
187 fprintf('Y\t\t');
188 fprintf('Y (hex)\t\t\t');
189 fprintf('Z\t\t');
190 fprintf('Z (hex)\t\t');
191 fprintf('delta\t\t');
192 fprintf('delta (hex)\t');
193 fprintf('mu\n');
194
195 for i = 1:n-m % print Phase 1 of the algorithm. 0 <= i< n-m
196     if shift(i) == 0
197     fprintf('%2d\t', i-1);
198     fprintf('%10d\t', y(i))
199     if y(i) < 0
200         y(i) = 2^n + y(i);
201     end
202     fprintf('%s\t', dec2hex(y(i),8));
203
204     fprintf('%10d\t\t', z(i))
205     if z(i) < 0
206         z(i) = 2^n + z(i);
207     end
208     fprintf('%s\t', dec2hex(z(i),8));
209
210     fprintf('%4d\t\t', delta(i))
211     fprintf('%s\t', dec2hex(delta(i),8));
212
213     fprintf('%2d\n', mu(i));
214     end
215
216 end
217 fprintf('-----------------------------------------------------\n\n')
218 fprintf('Output of Step 1:\n')
219
220 fprintf('\t%10d\t', y(n-m+1))
```

```
221 fprintf('%s\t', y_out);
222
223 fprintf('%10d\t\t', z(n-m+1))
224 fprintf('%s\n', z_out);
225
226
227
228 fprintf('-------------------------------------------------\n\n')
229 % Prepare y_step2 and z_step2 for printing HEX values
230 if y_step2 < 0
231       temp = 2^n + y_step2;
232 else
233      temp = y_step2;
234 end
235
236 y_step2_hex = dec2hex(temp,8);
237
238 if z_step2 < 0
239       temp = 2^n + z_step2;
240 else
241      temp = z_step2;
242 end
243 z_step2_hex = dec2hex(temp,8);
244
245 fprintf('Output of Step 2:\n')
246 fprintf('\t%10d\t', y_step2)
247 fprintf('%s\t', y_step2_hex);
248
249 fprintf('%10d\t\t', z_step2)
250 fprintf('%s\t', z_step2_hex);
251
252 fprintf('%4d\t\t', delta_step2)
253 fprintf('%s\t', dec2hex(delta_step2,8));
254
255 fprintf('%d\n', mu_step2);
```

A.2 2's complement function code

```
function x_bin = func_print_2complement(x,n)
% print a number x in 2's complement format
% using n bits


if x >= 0
    if x > 2^n-1
        fprintf('func_print_2complement:')
        fprintf('x = %d is bigger than 2^%d-1 = %d\n',x, n-1,
            2^(n-1)-1);
        return;
    end
    x_bin = dec2bin(x,n);
else
    if x < -2^(n-1)
        fprintf('func_print_2complement:')
        fprintf('x = %d is smaller than -2^%d = %d\n',x, n,
           -2^(n-1));
        return;
    end
    x = 2^n + x;
    x_bin = dec2bin(x+2^n,n);
end
```

A.3 Division function code

```
function [y_out,z_out, mu, mu_delta,shift_out] = ...
    func_divide_algorithm1(x,y_in,z_in,delta,i,shift)

mu = sign(y_in);

y_bin = func_print_2complement(y_in,32-i);
%y_bin = sscanf(dec2bin(y_in),'%1d');
```

```
8
9  %y_bin_shift = sscanf(dec2bin(bitshift(a,1)),'%1d')
10
11 if (xor(mu,bin2dec(y_bin(1)))) == 1
12     fprintf('No operation\n');
13     y_bin_shift = func_print_2complement((bitshift(y_in,1))
           ,32-i+1);
14     y_out_shift = y_bin_shift(1:end-1)
15     y_out = bin2dec(y_out_shift);
16    shift_out = 1;
17 else
18     y_out = y_in - mu*x*delta;
19     y_out_bin = dec2bin(y_out)
20     shift_out = 0;
21 end
22
23 %y_out = y_in - mu*x*delta;
24 if shift == 1
25     z_out = z_in - mu*delta;
26 else
27     fprintf('No shift\n');
28     z_out = z_in + mu*delta;
29 end
30
31
32 mu_delta = mu*delta;
```

Bibliography

[1] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and TheirApplications*. New York: Cambridge University Press, 1994.

[2] E. Mastrovito, *VLSI Architectures for Multiplication over Finite Field* $GF(2^m)$. Springer-Verlag, 1988.

[3] B.Sunar and C. Koç, "An efficient optimal normal basis type II multiplier," *IEEE Transactions on Computers*, pp. 83–87, Jan. 2001.

[4] J. Omura and J. Massey, "Computational method and apparatus for finite field arithmetic," May 1986.

[5] A. Reyhani-Masoleh and M. A. Hasan, "A new construction of massey-omura parallel multiplier over $GF(2^m)$," *IEEE Transactions on Computers*, pp. 511–520, 2002.

[6] L. Gao and G. E. Sobelman, "Improved VLSI designs for multiplication and inversion in $GF(2^m)$ over normal bases," in *In Proc. of the 13th Annual IEEE International ASIC/SOC Conference*, pp. 97–101, 2000.

[7] A. Reyhani-Masoleh and M. A. Hasan, "Low complexity word-level sequential normal basis multipliers," *IEEE Transactions on Computers*, pp. 98–110, 2005.

[8] A. Reyhani-Masoleh and M. A. Hasan, "Low complexity sequential normal basis multipliers over $GF(2^m)$," in *in Proc. of the 16th IEEE Symposium on Computer Arithmetic*, pp. 188–195, 2003.

[9] J.-S. Horng, I.-C. JOU, and C.-Y. Lee, "Low-complexity multiplexer-based normal basis multiplier over $GF(2^m)$," *Zhejiang University Science*, pp. 834–842, 2009.

[10] J. Blake, R. Roth, and G. Seroussi, *Efficient Arithmetic in $GF(2^n)$ through Palindromic Representation*. Hewlett-Packard, HPL-98-134, Aug. 1998.

[11] T. Beth and D. Gollman, "Algorithm engineering for public key algorithms," *IEEE J. Selected Areas in Comm.*, pp. 458–465, May 1989.

[12] C. W. Chiou, C.-Y. Lee, and Y.-C. Yeh, "Sequential type-1 optimal normal basis multiplier and multiplicative inverse in $GF(2^m)$," *Tamkang Journal of Science and Engineering*, pp. 423–432, 2010.

[13] C. C. Wang, T. K. Truong, H. M. Shao, L. J.Deutsch, J. K. Omura, and I. S. Reed, "VLSI architectures for computing multiplications and inverses in $GF(2^m)$," *IEEE Transactions on Computers*, pp. 709–717, Aug. 1985.

[14] J.-S. Horng, I.-C. Jou, and C.-Y. Lee, "On complexity of normal basis multiplier using modified booth's algorithm," in *in proc. of the 7th WSEAS International Conference on Applied Informatics and Communications*, (Athens,Greece), pp. 12–17, Aug. 2007.

[15] H. Li and C. N. Zhang, "Low-complexity versatile finite field multiplier in normal basis," *EURASIP Journal on Applied Signal Processing 9*, pp. 954–960, 2002.

[16] A. Reyhani-Masoleh and M. Hasan, "Efficient digit-serial normal basis multipliers over $GF(2^m)$," *ACM Trans. Embedded Computing Systems, special issue on embedded systems and security*, pp. 428–439, Apr. 2003.

[17] Y. Sukcho and J. Yeon Choi, "A new word-parallel bit-serial normal basis multiplier over $GF(2^m)$," *International Journal of control and Automation*, pp. 209–216, June 2013.

[18] A. Reyhani-Masoleh, "Efficient algorithms and architectures for field multiplication using gaussian normal bases," *IEEE Trans. Computers*, pp. 34–47, Jan. 2006.

[19] R. Azarderakhsh and A. Reyhani-Masoleh, "A modified low complexity digit-level gaussian normal basis multiplier," in *In proc. Third Int?l Workshop Arithmetic of Finite Fields (WAIFI)*, pp. 25–40, June 2010.

[20] N. I. of Standards and Technology, *Digital Signature Standard: FIPS Publication 186-4*. pub-NIST, Jan. 2000.

[21] F. Gebali and A. Ibrahim, "Systolic array architectures for sunar-koç optimal normal basis type II multiplier," *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI)SYSTEMS*, Oct. 2015.

[22] A. Menezes, ed., *Applications of Finite Fields*. Boston: Kluwer Academic, 1993.

[23] Y. Wang, Y. Cheung, and H. Liu, "A new parallel multiplier for type II optimal normal basis," *IEEE Transactions on Computers*, pp. 460–469, Jan. 2007.

[24] F. Gebali, *Algorithms and Parallel Computing*. Wiley, Apr. 2011.

[25] H. T. Kung, "Why systolic architectures?.," *IEEE Trans. Computers*, pp. 37–46, 1982.