Unified Field Multiplier For GF(p) and GF(2^n) with Novel Digit Encoding

Thesis by

Lai Sze Au

In Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

Cardiff University

Cardiff School of Engineering
PO Box 935, Cardiff, CF24 0YF
Wales, UK

2004

(Submitted October 30, 2004)

DECLARATION

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree

Signed..(Candidate)

Date.......20/10/04..

STATEMENT 1

This thesis is the result of my own investigations, except where otherwise stated.

Other sources are acknowledged by footnotes giving explicit references. A

Bibliography is appended.

Signed..(Candidate)

Date.........30/10/04..

STATEMENT 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed..(Candidate)

Date...31/10/04..

Acknowledgement

I would like to take this opportunity, first of all, to say thank you to my supervisor, Prof. N. Burgess. Without his guidance and support, none of these would have been possible. He has always been so patient with me and I have truly learnt a lot from him.

Secondly, I would like to thank ARM Ltd. for sponsoring me. I would also like to express my appreciation to my parents, my brother and sisters for being so supportive.

Finally, big THANK YOU to my husband! Thank you for everything!

ABSTRACT

In recent years, there has been an increase in demand for unified field multipliers for Elliptic Curve Cryptography in the electronics industry because they provide flexibility for customers to choose between Prime (GF(p)) and Binary (GF(2^n)) Galois Fields. Also, having the ability to carry out arithmetic over both GF(p) and GF(2^n) in the same hardware provides the possibility of performing any cryptographic operation that requires the use of both fields. The unified field multiplier is relatively future proof compared with multipliers that only perform arithmetic over a single chosen field. The security provided by the architecture is also very important. It is known that the longer the key length, the more susceptible the system is to differential power attacks due to the increased amount of data leakage. Therefore, it is beneficial to design hardware that is scalable, so that more data can be processed per cycle. Another advantage of designing a multiplier that is capable of dealing with long word length is improvement in performance in terms of delay, because less cycles are needed. This is very important because typical elliptic curve cryptography involves key size of 160 bits.

A novel unified field radix-4 multiplier using Montgomery Multiplication for the use of GF(p) and GF(2^n) has been proposed. This design makes use of the unexploited state in number representation for operation in GF(2^n) where all carries are suppressed. The addition is carried out using a modified (4:2) redundant adder to accommodate the extra 1* state. The proposed adder and the partial product generator design are capable of radix-4 operation, which reduces the number of computation cycles required. Also, the proposed adder is more scalable than existing designs.

Contents

FIGURES

TABLES

List of Abbreviations

GF(p) / F_p	Prime Galois Field
GF(2^n) / F_2^n	Binary Galois Field
DES	Data Encryption Standard
DEA	Data Encryption Algorithm
IDEA	International Data Encryption Algorithm
PES	Proposed Encryption Standard
IPES	Improved Proposed Encryption Standard
AES	Advanced Encryption Standard
mod	Modular
SHA	Secure hash Algorithm
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECDSA	Elliptic Curve Digital Signature Standard
ECDH	Elliptic Curve Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
$\oplus$	XOR
DPA	Differential Power Analysis
$\wedge$	AND logic
$\vee$	OR logic
FO4	Fan-out of 4
Logical Effort: g	Logical effort
Logical Effort: b	Branching effort
Logical Effort: h	Electrical effort
Logical Effort: p	Parasitic effort
Logical Effort: N	Number of Stages
Logical Effort: F	Path Effort
Logical Effort: D	Delay
div	Divide (Arithmetic function)
$\hat{q}$	Estimated q
gcd	Greatest Common Denominator
PPG	Partial Product Generator
PP	Partial Product
PS	Partial Sum
M_i	i^{th} bit of the modulus M
R	Remainder

w	Wordlength
C_i	i^{th} bit of the carry C
FSEL	Galois Field Selection
MUX	Multiplexer
$\boxplus$	Addition mod 2^{16}
$\boxed{\triangle}$	register
X	Discard bit
▬	Pipeline latches
$\boxdot$	PPG Generation

1 Introduction

1.1 Motivation

This thesis will describe the VLSI implementation of a modular multiplier that is capable of performing multiplication in both GF(p) and GF(2^n) Galois Fields for elliptic curve cryptography. Elliptic curve cryptography is becoming more popular compared with traditional cryptographic systems because it provides a similar level of security but much smaller key lengths are required.

One of the most used operations in cryptographic systems is modular exponentiation, which involves many long wordlength modular multiplications. For elliptic curve cryptography, modular multiplication is one of the most computationally demanding operations involved. For these reasons, elliptic curve cryptography was chosen to be the target system and hardware implementation for modular multiplication will be designed.

In recent years, there has been an increase in demand for unified field multipliers for Elliptic Curve Cryptography in the electronics industry, because they provide flexibility for the customer to choose between the Prime (GF(p)) and the Binary (GF(2^n)) Field. Also, having the ability to carry out arithmetic over both GF(p) and GF(2^n) in the same hardware provides the possibility of performing any cryptographic operation that requires the use of both fields. The unified field multiplier is relatively future proof compared with multipliers that only perform arithmetic over a single chosen field.

The security provided by the architecture is also very important. It is known that the longer the key length, the more susceptible the system is to differential power attacks, due to the increase amount of data leakage. Therefore, it is beneficial to design a hardware that is scalable, so that more data can be processed per cycle. A scalable system is a system that can expand word length without affecting logic depth. An additional advantage of designing a multiplier that is capable of dealing with long wordlengths is improvement in performance in terms of delay, as less iterations are needed. This is very important because typical elliptic curve cryptography involves key size of 160 bits.

Apart from being scalable and capable of dual field operation, the system must also be impartial, which means that it must not favour either of the fields. This unified field multiplier should avoid the need to compromise on speed and area in order to gain the dual field ability.

1.2 Thesis Outline

In this thesis, the hardware implementation of ECC in two different fields GF(p) and GF(2^n) will be explored. Even though GF(p) and GF(2^n) are structurally very different, they are very similar in nature, this can be exploited when designing this unified multiplier. This unified ECC multiplier for GF(p) and GF(2^n) provides a simple generic solution to the industry which could give flexibility to their customers to choose between GF(p) and GF(2^n) field with minimal penalty. Note that all the arithmetic operations in systems such as AES are carried out in finite field GF(2^n), therefore the proposed multiplier could be used for this as well.

Unlike the existing designs, the proposed design does not require an external control signal that will be propagated to all the cell modules, which cause very high fan-out to the field selection signal that could affect the scalability of the design. Instead, the proposed design makes use of a unique 1* implementation to embed field information into the number encoding itself.

The proposed multiplier will operate in radix-4; by increasing the radix of the system, the number of iterations will be reduced. However, there is a trade-off between the area consumption and the improvement in speed. Radix-4 system is considered to have the best trade-off between speed and area.

Furthermore, the design uses redundant addition to avoid long carry chains. The multiplier is in digit serial fashion and Montgomery multiplication is used.

The thesis is organised as follows:

Chapter 2 will provides the basic background theory on cryptography, particularly on common systems such as DES and RSA. Elliptic curve cryptography systems are also introduced.

In Chapter 3, the operational details of elliptic curves functions will be explored. This will first be explained in basic real number groups, then elliptic curve groups over GF(p) and GF(2^n) will be investigated. The second part of Chapter 3 will be dedicated to explaining side channel attacks, e.g. differential power attacks and timing attacks. The final section of Chapter 3 will explore what other unified field operators in particular multipliers have been implemented.

Chapter 4 describes the implementation of the proposed unified field redundant adder for multiplication in either GF(p) or GF(2^n). This section will explain the unique encoding method that is employed in this design. The scalability of the adder will be assessed at the end of the chapter using a technique called Logical Effort.

Chapter 5 will present the overall implementation of the dual field multiplier. Montgomery multiplication will be compared with other modular multiplication methods such as Barrett modular multiplication, showing that Montgomery is more appropriate for the design. It will show the implementation of the partial product generator, modular reduction, carry test unit and the final redundant to binary conversion unit.

Chapter 6 will assess the results of the multiplier, suggest alternative methods to improve the operation of the multiplier, such as using a 4-input tri-state inverter to implement a multiplexer, rather than using a traditional design. This chapter will also examine radix-2 multiplier design and investigate the operation of the multiplier for $M\times\bar{M}$ multiplication. Finally, there is a conclusion.

2 Cryptography

Cryptography has become an integral part of modern day life as it provides secure communications. It provides a set of techniques to achieve the goal of different aspects of security, which are: confidentiality; data integrity; authentication and non-repudiation. Therefore, in order to achieve an adequate level of security for any communication, the following issues should be addressed:

- The data should only be readable by authorised recipients;
- the data itself should not be altered by any unauthorised person;
- parties involved in the communication and the data origination should be identified and authenticated;
- finally, no actions taken by any party involved are deniable.

This is achieved by encrypting the original data, or the plaintext, with a mathematical function, in order to convert it to the ciphertext. The recipient will then decrypt the ciphertext with a mathematical function to obtain the plaintext. It is assumed that the transmission medium is unsecured and eavesdropper could interrupt, intercept, modify or fabricate the data. Figure 2.1 shows the basic concept of encryption and decryption. The plaintext m is encrypted by function E with key e. The ciphertext C is then transmitted to the recipient. The recipient decrypts C using function D with key d and the decrypted text can be retrieved. The keys should possess the following properties: $D_d = E_e^{-1}$, therefore, $D_d(E_e(m)) = m$. The two keys d and e could be the same.

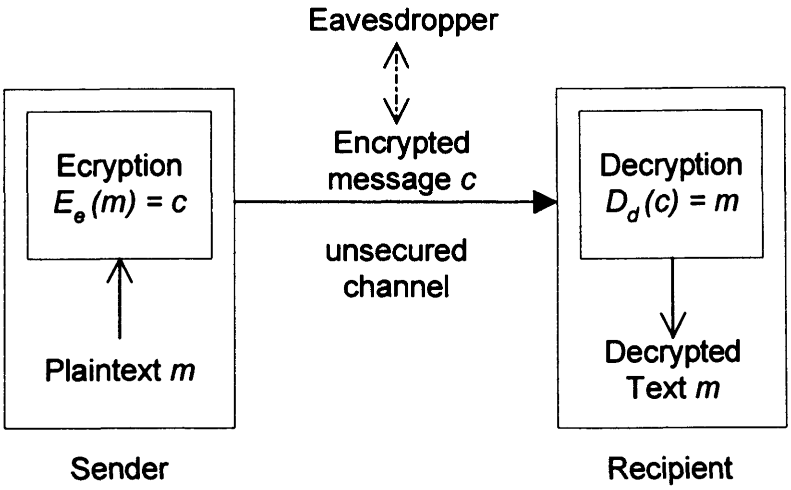


Figure 2.1 Schematic to show encryption and decryption

The security of a cryptosystem depends upon the strength of the algorithm, and the length of the keys used for encryption and decryption. A cryptosystem should be secure enough to be able to avoid most attacks except "brute-force", which is a known-plaintext attack. The key length must be sufficiently great so that brute-force attack becomes computationally infeasible. However, it has been discovered that the longer the key is required by a system, the higher chance of the system being successfully attacked by differential power attacks [1]. Differential power attacks will be discussed in Section 3.4.4.

One-way function and trapdoor-one-way-function form the backbone of modern (public key) cryptography. One-way function is relatively easy to carry out in one direction, but computationally infeasible or impossible to carry out the reverse operation. Trapdoor-one-way-function is a one-way function such that, given the extra trapdoor information, the computation of the reverse function becomes computationally feasible or else it will be difficult or even impossible to reverse.

The main aim of cryptography is to keep the communication secure, which fundamentally means keeping the plaintext secret by keeping the decryption key secret. Cryptanalysis is the science of recovering the plaintext of the message without the knowledge of the key. Each attempt at Cryptanalysis is called an "attack". One should assume that the cryptanalyst has complete details of the cryptographic algorithm and implementation. There are six commonly known attacks on encryption schemes rather than attacks on the implementation.

1. A *ciphertext-only attack* is one where the cryptanalyst only has knowledge of the ciphertext and tries to deduce the plaintext or the decryption key by observing the ciphertext.

2. A *known-plaintext attack* is one where the cryptanalyst obtains a quantity of plaintext and its corresponding ciphertext.

3. A *chosen-plaintext attack* is one where the cryptanalyst chooses the plaintext and is then able to obtain the corresponding ciphertext, by using this information the cryptanalyst can deduce the remaining plaintext of the ciphertext which was previously unseen.

4. An *adaptive chosen-plaintext attack* is fundamentally a chosen-plaintext attack, however, the choice of plaintext may depend on the ciphertext from the previous requests.

5. A *chosen-ciphertext attack* is one where the cryptanalyst chooses the ciphertext and is then able to obtain the corresponding plaintext. However, in order to carry out such operation the attacker needs to be able to gain access to the decryption equipment but not the decryption key. The cryptanalyst could now recover the corresponding plaintext of the different ciphertext from the information deduced before without access to such equipment.

6. An *adaptive chosen-ciphertext attack* is fundamentally a chosen-ciphertext attack, however, the choice of ciphertext may depend on the plaintext from the previous requests.

There are two different types of cryptography, Symmetric (Private Key) and Asymmetric (Public Key) cryptography.

2.1 Symmetric key cryptography

Symmetric key cryptography is a cryptographic system where the encryption key (*e*) is generated from the decryption key (*d*) or vice versa, as shown in the following equation:

$$d = e^{-1} \tag{2.1}$$

In most cases, the most practical choice of the key pair for symmetric key cryptography is when the encryption key (*e*) equals to the decryption key (*d*). Therefore, in order to keep the data secure, the key used must be kept secret. As a result, symmetric key cryptography is also called private key cryptography. However, there lies a so-called "key distribution problem", where an efficient protocol needs to be established for key agreement and key exchange in a secure manner (such protocols will not be discussed in this thesis). Figure 2.2 shows the schematic diagram of symmetric key cryptography.

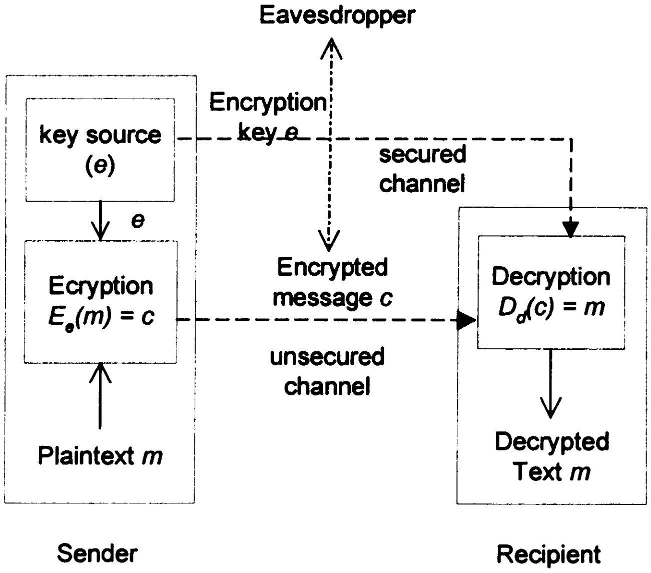


Figure 2.2 Schematic to show Symmetric key cryptography

There are two different types of symmetric key cryptography, and they are block cipher and stream cipher. Block ciphers can be either symmetric key or public key; only symmetric key block ciphers are addressed in this section.

Block ciphers operate on plaintext and ciphertext in a group of bits, usually 64 bits but it could be longer. The blocklength of the plaintext blocks and the ciphertext blocks are the same. The most important classes of block ciphers are substitution ciphers and transposition ciphers. These ciphers substitute symbols or a group of symbols by other symbols or other group of symbols or the symbols in a block is permuted by a transposition function.

The encryption transformation E_e of a simple substitution cipher where $e \in K$ can be shown as follows:

$$E_e(m) = (e(m_1)e(m_2)...e(m_t)) = (c_1c_2...c_t) = c, \qquad (2.2)$$

over A, defined to be an alphabet of q symbols, and M, the set of all strings of length t over A. In addition, K is the set of all permutations on the set A. $m = (m_1m_2...m_t) \in M$ For decryption, inverse permutation is carried out such that $d = e^{-1}$. The decryption transformation is:

$$D_d(c) = (d(c_1)d(c_2)...d(c_t)) = (m_1m_2...m_t) = m. \qquad (2.3)$$

For an encryption scheme with block length t, the encryption function e which is a transposition function, where K is defined as the set of all permutation on the set $\{1, 2...t\}$ can be shown as follows:

$$E_e(m) = (m_{e(1)}m_{e(2)}...m_{e(t)}) = (c_1c_2...c_t) = c. \qquad (2.4)$$

For decryption, inverse permutation is carried out such that $d = e^{-1}$. The decryption function is:

$$D_d(c) = (c_{d(1)}c_{d(2)}...c_{d(t)}) = (m_1m_2...m_t) = m \qquad (2.5)$$

Stream ciphers operate on streams of plaintext and ciphertext. They convert plaintext to ciphertext one bit at a time. Therefore, they can be considered as a very simple block cipher with block length equal to one. The encryption transformation E_e could be defined as follows, which makes use of a simple substitution cipher with block length equals to one:

$$c_i = E_{ei}(m_i), \qquad (2.6)$$

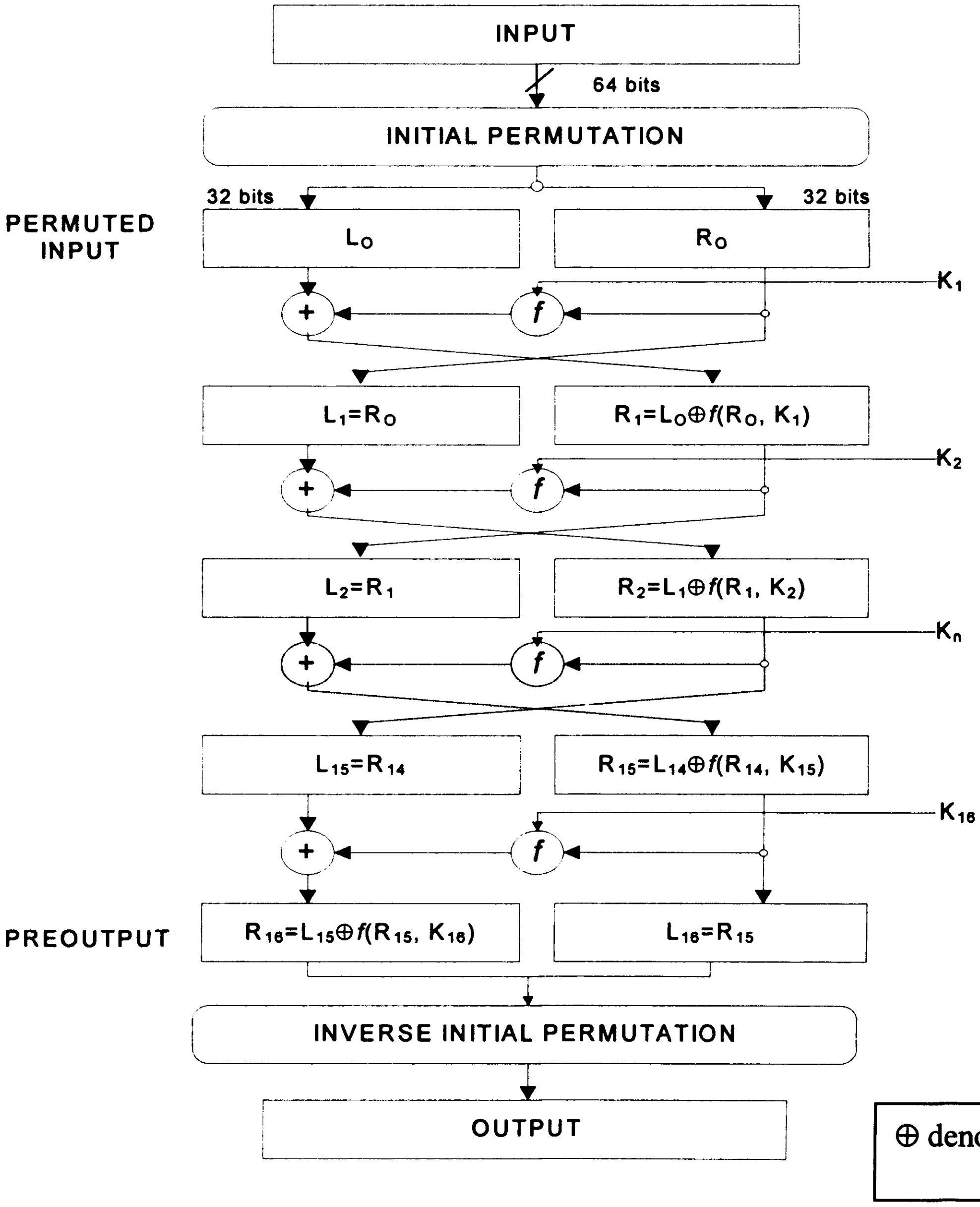
The keystream is defined as a sequence of symbols $e_1e_2e_3...e_i \in K$, where K is the key space for a set of encryption transformations. The keystream can be generated at random by an algorithm called keystream generator. The decryption transformation can be described as follows, where d_i denotes the inverse of e_i:

$$m_i = D_{di}(c_i). \qquad (2.7)$$

In general, block ciphers are more suitable for software implementations and stream ciphers are faster and require less complex implementation in hardware. Stream ciphers are more appropriate for situations where buffering is limited and error transmissions are high because they have little or no error propagation.

2.2 DES and Triple DES

The most popular symmetric key cryptographic algorithm is Data Encryption Standard (DES). DES is symmetric block cipher that uses 56-bit encryption key and has 64-bit block size. This is essentially an improvement of the algorithm "Lucifer" developed by IBM in the early 1970s [2]. In July 1977, the National Institute of Standards and Technology (NIST) adopted and issued DES as Federal Information Processing Standard Publication 46 (FIBS PUB 46) [3]. It provided standards and guidelines for this algorithm to be used by US Federal agencies. However, the standard could also be implemented and used by those outside the Federal government, such as for commercial use. The American National Standards Institute (ANSI) approved DES as a voluntary standard in 1981 (ANSI X3.92) [4], calling it the Data Encryption Algorithm (DEA). Figure 2.3 shows the encryption process of DES.


INPUT
64 bits
INITIAL PERMUTATION
32 bits
32 bits
PERMUTED INPUT
L_0
R_0
K_1
+
f
$L_1=R_0$
$R_1=L_0\oplus f(R_0, K_1)$
K_2
$L_2=R_1$
$R_2=L_1\oplus f(R_1, K_2)$
K_n
$L_{15}=R_{14}$
$R_{15}=L_{14}\oplus f(R_{14}, K_{15})$
K_{16}
PREOUTPUT
$R_{16}=L_{15}\oplus f(R_{15}, K_{16})$
$L_{16}=R_{15}$
INVERSE INITIAL PERMUTATION
OUTPUT

⊕ denotes XOR

Figure 2.3 DES Encryption

DES is a Feistel network, which means that a block of length *n* is divided into two halves: *L* and *R*. An iterated block cipher is defined where the outputs of the i^{th} round is determined from the output of the previous round and are defined as follows:

$$L_i = R_{i-1} \tag{2.8}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \tag{2.9}$$

where K$_i$ is the sub-key for i^{th} iteration and *f* is an arbitrary function. Feistel function is invertible and so the same algorithm can be used for both encryption and decryption.

For DES encryption, the 64 bits input block will first undergo the initial permutation *IP*. The permutated input block becomes the input of a 16-stage complex key computation process. The output of the 16-stage computation plus a final stage of block interchange produce the preoutput block, which will then undergo the inverse of the initial permutation to provide the final encrypted output. The cipher function *f* operates on two blocks, the 32 bits R_n and the 48 bits K_n chosen from the 64-bit key and produces a block of 32 bits. This 32 bits block becomes the block *R* input for the next iteration. Figure 2.4 depicts the DES cipher function *f* and the function is described as follows:

$$f(R_{i-1}, K_i) = P(S(E(R_{i-1} \oplus K_i))), \tag{2.10}$$

The implementation of DES is relatively easy particularly on special purpose chip due to its repetitive nature.

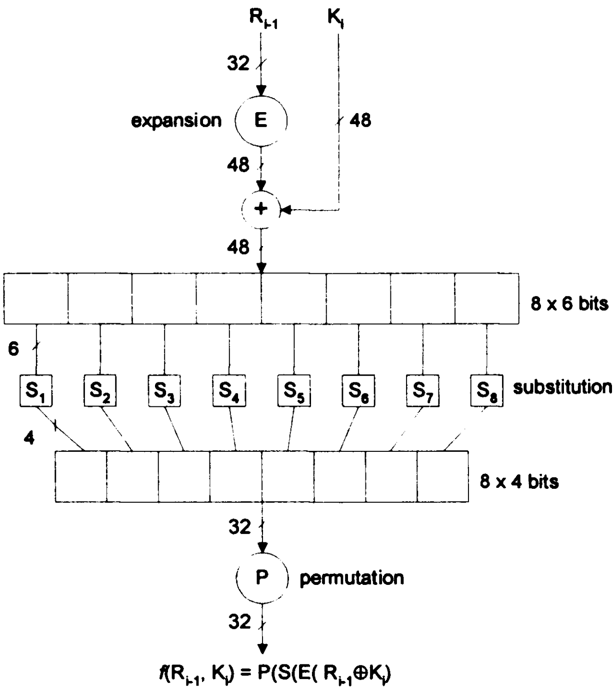


Figure 2.4 DES inner function ***f***

However, due to improvement in computational power, "single" 56-bit key DES has become less secure and so "single" DES has been phased out and is being replaced by Triple DES algorithm (TDEA) [5], which has an effective key length of 156 bits, even though the overall key length is 192 bits. This is because although the input key of DES is 64 bits, only 56 bits are actually used by the DES algorithm. The other 8 bits, which are not used by the algorithm, may be used for error detection and the least significant bit in each byte is a parity bit and is ignored. The procedure for encryption is the same as regular DES, but it is repeated three times and so TDEA is three times slower than single DES. The input block is first encrypted using DES algorithm with the first key (KEY1), the output will then become the input of the second stage where the block will be decrypted using the second key (KEY2). Finally, the output of the second stage will become the input of the third and last stage. The input block is encrypted using the third key (KEY3). The final result will then be used in the computation of the ciphertext. None of the intermediate results is revealed outside the cryptographic boundary.

DES is a hardware friendly algorithm due to its regular structure, which makes exploitation of parallelism by pipelining easy. DES and Triple-DES commercial implementations are generally available in smart card IC design, such as mifare® pro X P8RF6008 by Philips [6]. This particular design makes use of a Triple-DES co-processor to speed up the calculation time. According to the specifications, by using a co-processor, about three orders of magnitude of speed improvement could be achieved compared to software solutions and the total time for a triple-DES calculation to less than 35μs at 13.56MHz. Another example of Triple-DES IC smart card design is KS88C92008/4/2/1 by Samsung Electronics [7]. In this implementation, a specific Triple-DES module is included in the smart card design. The fastest DES execution time performed by this module is 77.9 μs at 4.52 MHz.

Even though DES was developed for implementation on hardware, many software implementations have been developed. However, DES implementation in software tends to be less efficient, an example of software implementation of DES can be found in [8]. In this report, the author shows the results of DES algorithm implementation using C code on the C6000 Digital Signal Processing platform and compiled on the Texas Instruments' Optimizing C Compiler where no assembly code is used. In this report, DES is implemented at data rates as high as 52.4 Mbits per second for DES and 22.3 Mbits per second for triple-DES on the C6201 McEVM (200 MHz). Using the C6211 DSK (150 MHz), data rates were measured as high as 38.8 Mbits per second for DES and 17.8 Mbits per second for triple-DES. The author found that using C code for DES implementation provides flexibility, and is a quick and inexpensive way to add encryption functionality to a design. Other DES software implementation examples can be found in [161] and [163].

2.3 Other Symmetrical Block Cipher Algorithm – IDEA & AES

In this section more symmetrical block cipher algorithm are introduced and they are IDEA and AES.

2.3.1 IDEA

International Data Encryption Algorithm (IDEA) is a symmetrical block cipher algorithm with a 64-bit block length and a 128-bit input key. It was first proposed in 1990 by Xuejia Lai and James Massey and was called Proposed Encryption Standard (PES) [9]. It was then strengthened and renamed, from Improved Proposed Encryption Standard (IPES) to IDEA in 1992 [10]. Figure 2.5 shows the clock diagram of the IDEA algorithm. As in DES, IDEA is of the Feistel structure, it consists of eight iterations plus an output transformation only. The 64-bit input block is divided into four 16-bit sub-blocks and are denoted as X_1, X_2, X_3, X_4. In the course of each iteration, the following events will take place:

1. Multiply plain text block X_1 by the first sub-key K_1

2. Add plain text block X_2 with the second sub-key K_2

3. Add plain text block X_3 with the third sub-key K_3

4. Multiply plain text block X_4 by the fourth sub-key K_4

5. XOR the results of step 1 and step 3

6. XOR the results of step 2 and step 4

7. Multiply the result of step 5 by the fifth sub-key K_5

8. Add the results of step 6 and step 7

9. Multiply the result of step 8 by the sixth sub-key K_6

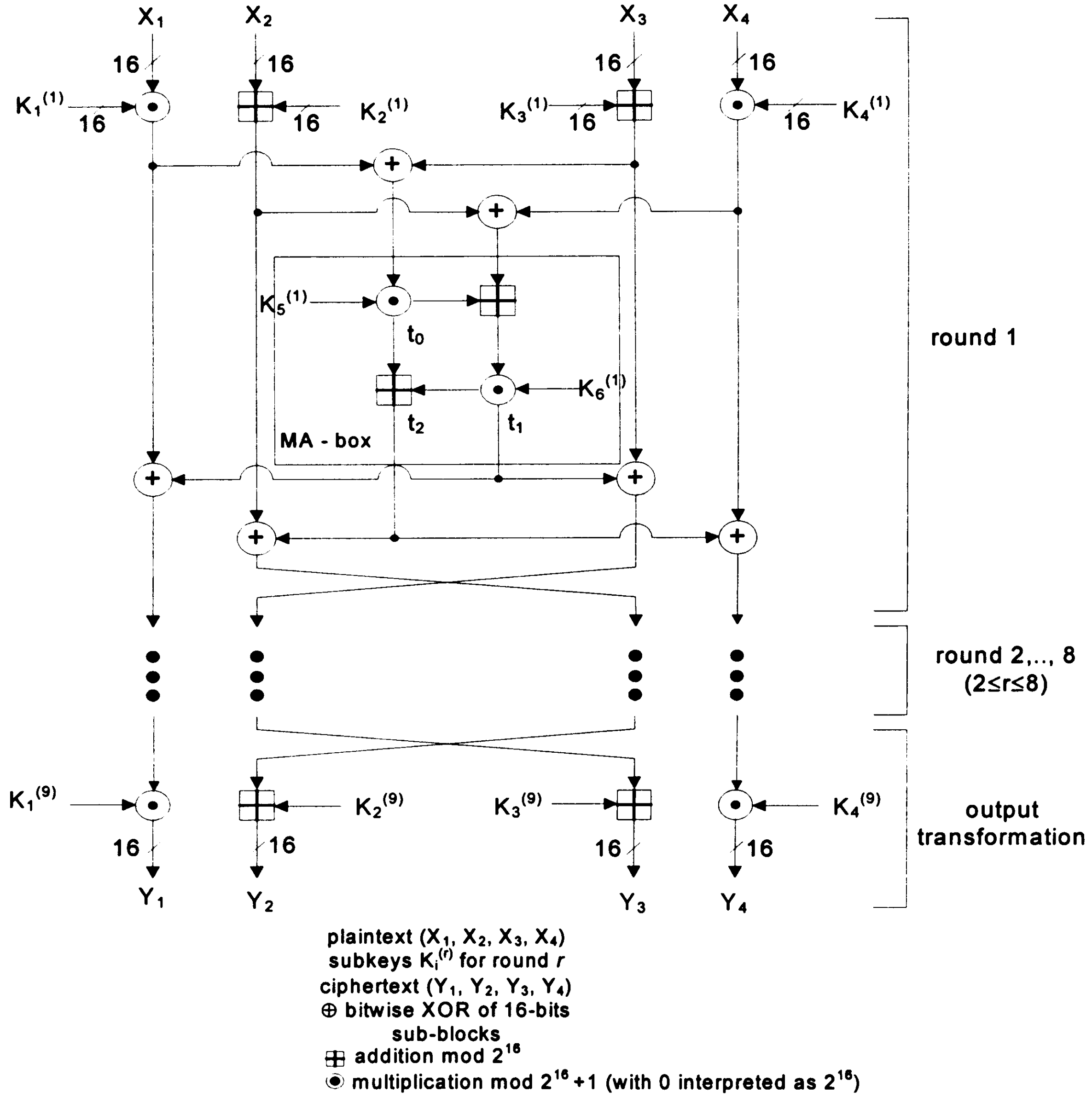
10. Add the results of step 7 and step 9

11. XOR the results of step 1 and step 9

12. XOR the results of step 3 and step 9

13. XOR the results of step 2 and step 10

14. XOR the results of step 4 and step 10



X1 X2 X3 X4
16
K1(1) K2(1) K3(1) K4(1)
K5(1)
K6(1)
t0 t1 t2
MA - box
round 1
round 2,..., 8
(2≤r≤8)
K1(9) K2(9) K3(9) K4(9)
output
transformation
Y1 Y2 Y3 Y4
plaintext (X1, X2, X3, X4)
subkeys Ki(r) for round r
ciphertext (Y1, Y2, Y3, Y4)
⊕ bitwise XOR of 16-bits sub-blocks
⊞ addition mod 2^16
⊙ multiplication mod 2^16+1 (with 0 interpreted as 2^16)

Figure 2.5 Block Diagram of the IDEA algorithm

2.3.2 AES

In September 1997, NIST initiated the development of a new Encryption Standard – Advanced Encryption Standard (AES), they requested candidates develop a new algorithm strong enough to replace DES and Triple DES. On 26th November, 2001, NIST published a new federal standard known as FIPS PUB 197 [11], the chosen algorithm was called Rijndael [12].

The AES algorithm is a symmetric block cipher that is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. AES processes data in a group of eight (a byte), i.e., a sequence of eight bits is treated as one entity. Therefore, all the arithmetic operations required are carried out in finite field GF(2^8). More details on arithmetic operations in GF(2^8) are described in chapter 3.

2.4 Public Key Cryptography

As mentioned in section 2.1, secret key cryptography has a weakness of "key distribution problem" or "key management problem". This is caused by the fact that the same key is used for both encryption and decryption and therefore the key must be kept secret. Public key cryptography or asymmetric key cryptography overcomes this problem by using different encryption (*e*) and decryption (*d*) key pair so that confidentiality, data integrity, authentication and non-repudiation can be achieved. Figure 2.6 shows the schematic diagram of public key cryptography operations.

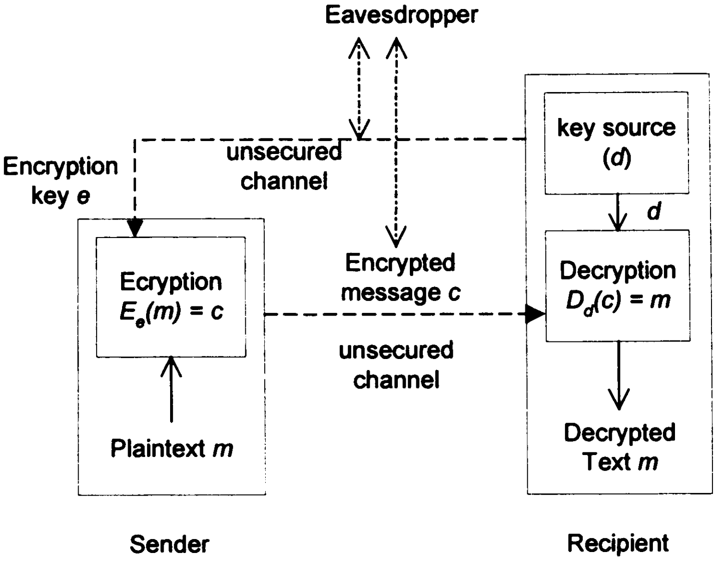


Figure 2.6 Schematic to show public key cryptography

2.4.1 Diffie-Hellman key agreement protocol

In 1976, Diffie and Hellman proposed a new cryptography system that makes use of public key with a key agreement protocol [13], which provided the ground for future development of public key cryptography. The key agreement protocol provides a means for two users to agree on a secret key over an insecure channel without prior arrangement. The security of the protocol depends on the discrete logarithm problem over a finite field GF(p) where p is a prime number. It has been proven that under certain assumptions breaking the Diffie-Hellman protocol is equivalent to computing discrete logarithms [14]. The Diffie-Hellman key agreement protocol involves a prime parameter p and a generator g, which is an integer smaller than p. Let:

$$n = g^k \bmod p \quad \text{for } 1 \le k \le p\text{-}1 \tag{2.11}$$

Therefore, k is:

$$k = \log_g n \bmod p \quad \text{for } 1 \le n \le p\text{-}1 \tag{2.12}$$

User A and user B can share a secret key by first of all generating a random private value a and b respectively, where both value a and b are from the set of integers {1, ..., p-1}. They then generate their respective public keys:

$$g^a \bmod p \text{ and } g^b \bmod p \tag{2.13}$$

When the two users wish to communicate privately, they exchange their public keys.

A computes:

$$g^{ab} = (g^b)^a \bmod p \tag{2.14}$$

User B computes:

$$g^{ba} = (g^a)^b \bmod p \tag{2.15}$$

Since $g^{ab} = g^{ba}$, therefore they now have a shared key. It is assumed that the secret component g^{ab} is computationally infeasible to be worked out providing the random prime number p is large enough. In the same paper, Diffie and Hellman also suggested the idea of using digital signatures to ensure the authenticity of the data. Much research has been done to improve the authenticity of the data, Diffee *et al* proposed an authentication and authenticated key exchange protocol called station-to-station (STS) in 1992 [15].

Public key cryptography works on the assumption that it is computationally infeasible to work out decryption key d given the encryption key e. Encryption E_e is being viewed here as a trapdoor one-way function with the decryption key d being the trapdoor information necessary to compute the inverse function, hence allow decryption. Given that it is not necessary for the encryption key e to be kept secret, therefore it can be made public and hence it is called the public key. However, the corresponding decryption key must be kept secret and hence it is called the private key. The public key allows any entity to send encrypted messages to the same recipient using the same public key, only the intended recipient can decrypt the message using their private key.

It is particularly important for public key cryptography to ensure the authenticity of the public keys to avoid protocol failure where the origin of the public keys are not known. Without appropriate measures, an adversary could impersonate the intended recipient B and issue false public key to sender A. The adversary will then be able to intercept and decrypt the message with the private key before sending the message to B, which is now encrypted with B's public key. One solution to such problem is by making use of Digital Signature Algorithm (DSA). The concept of digital signature algorithm is as follows:

Sender A "signs" the message set $M = \{m_1, m_2, m_3, ...\}$ by using a signature transformation function S_A, which will be kept secret by the sender. The signature transformation function transforms the message set to give signature set $S = \{s_1, s_2, s_3, ...\}$. This transformation can be interpreted as follow:

$$s = S_A(m) \tag{2.16}$$

Sender A transmits the signature pair (m; s) where s is the signature for message m. The recipient can verify the signature by obtaining the verification function V_A from A. Recipient B computes the following to verify the signature:

$$u = V_A(m;s) \tag{2.17}$$

The recipient accepts the signature when the signature pair (m; s) matches; otherwise, the signature is rejected.

In general, DSA enables digital signatures to be generated and verified. It is a pair of large numbers, which form the key pair, that are computed according to the specified algorithm, within parameters to verify the authenticity of the signature and hence the integrity of the data.

In practice, a hash function, sometimes called one-way hash function, is used in the signature generation process to obtain a condensed version of data called a message digest. Figure 2.7 shows digital signature generation and verification with a Secure Hash Algorithm. The Secure Hash Standard that is specified by NIST is known as FIPS PUB 180-1 [16] and the Secure Hash Algorithm is called SHA-1.

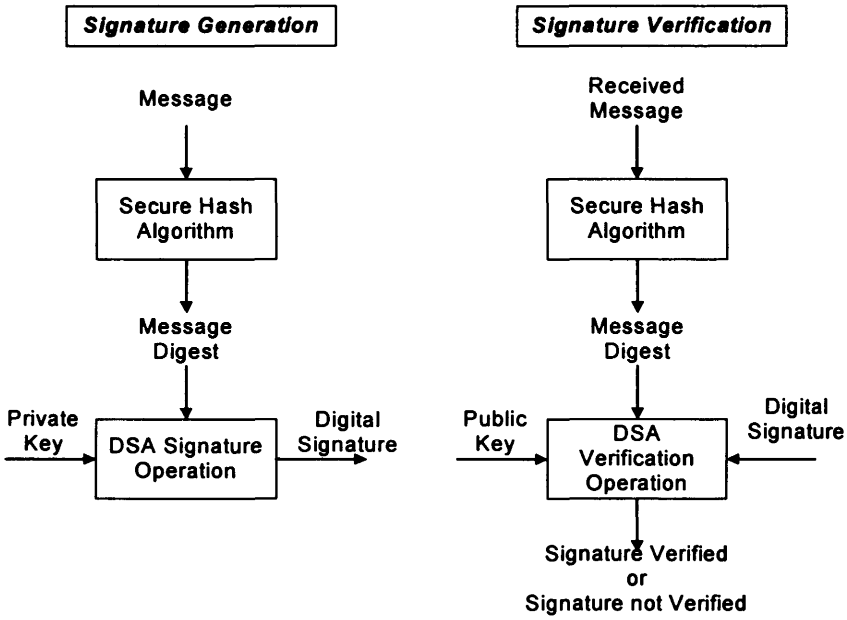


Figure 2.7 DSA with SHA

The hash function is a process that produces a condensed version of data of arbitrary length (signed or pre-signed message), called the hash value. In the case of FIPS 180-1 standard, the input message must be less than 2^{64} bits in length, the output of the Secure Hash algorithm is called the message digest, and is 160 bits long. Hash function has the following properties:

1. It is relatively easy to compute for any given input.

2. It is one-way.

3. It is collision-free – this means that the Hash function is computationally infeasible to find any two messages *x* and *y* such that *H(x) = H(y)*.

For the digital signature generation process, the message digest becomes one of the two inputs to the digital signature operation, which generates the digital signature as the output. The digital signature and the signed message are sent to the verifier.

For digital signature verification process, the message digest is one of the three inputs to the digital signature verification operation, which verifies whether the signature matches or not. The Hash function used must be the same as the one used for signature generation.

Note that for digital signature operation with signature hash algorithm, only the hash value is signed. Compared with processes where the message is signed directly, signing just the message digest saves time and space. With the direct signing method, the message needs to be split into blocks of appropriate size and each block is signed individually.

Various other signature schemes exist, such as, the ElGamal signature scheme [17] and the Digital Signature Standard (DSS) FIPS 186 published by NIST in 1991 [18], which is a variant of ElGamal scheme. Another well-known digital signature scheme is RSA signature scheme, which was first introduced in 1977 [19] [20]. In July 2002, Alfred Menezes published Evaluation of Security Level of Cryptography: RSA Signature Schemes (PKCS#1 v1.5, ANSI X9.31, ISO 9796), which gave a good overview of the security of RSA signature schemes [21]. Digital Signature Standard (DSS) published by NIST on 27 January 2000 [22] described three algorithms for digital signature generation and verification. They are the following:

1. Digital Signature Algorithm (DSA)
2. RSA digital signature algorithm
3. Elliptic Curve Digital Signature Algorithm (ECDSA)

2.4.2 RSA

RSA cryptosystem was invented by R. Rivest, A. Shamir, and L. Adleman in 1978 [20], and is one of the most widely used public key cryptosystems. As mentioned previously, RSA not only encrypts and decrypts messages; it can also be used for digital signatures. Its strength is based on integer factorization problem, where a large number is to be factorized.

Prior to encryption, the RSA public key and the corresponding private key have to be generated before the entity A could encrypt the message with entity B's public key. The procedure for RSA key generation is as follows:

1. Generate two large random and distinct prime numbers p and q, for maximum security, p and q should be of equal length.

2. Compute the product:

   $$n = pq \tag{2.18}$$

   and

   $$\varnothing = (p - 1)(q - 1) \tag{2.19}$$

3. Select a random integer encryption exponent e, $1 < e < \varnothing$, such that gcd(e, $\varnothing$) = 1, i.e. they are relatively prime.

4. Use the extended Euclidean algorithm (*see Appendix 1*) to compute the unique integer decryption exponent d, $1 < d < \varnothing$, such that $ed = 1 \bmod \varnothing$, hence, d and n are also relatively prime.

   $$d = e^{-1} \bmod ((p-1)(q-1)) \tag{2.20}$$

5. B's public key is (n, e); B's private key is d.

6. p and q could now be discarded and should never be revealed.

A can now encrypt a message for *B*, and *B* can decrypt using the private key. The RSA encryption and decryption procedures are as follows:

1. Encryption:

   a. Obtain *B*'s authentic public key (n, e).

   b. Represent the message as an integer m in the interval [0; n - 1].

   c. Compute $c = m^e \bmod n$

   d. Send the ciphertext c to *B*.

2. Decryption:

   A decrypts plaintext m from c by carrying out the following:

   $$m = c^d \bmod n.$$

The assumption is that the RSA function is a trapdoor one-way function and the private key is the trapdoor. In order to compute $c = m^e \bmod n$ efficiently, a "Repeated square-and-multiply algorithm for exponentiation in Z_n" could be used (*see Appendix 1*). This kind of modular exponentiation is performed each time a part of the message is encrypted/decrypted. Both e and n are very large integers and so this operation is very computationally intensive, however, the Chinese Remainder Theorem (CRT) (*see Appendix 1*) can be used as a method for computing the modular exponentiation. By using CRT, the large modulo exponentiation can be split into two smaller exponentiations, namely over p and over q, which are already known. Fermat's Little Theorem (*see Appendix 1*) can be used to further reduce the size of the problem.

Even with these improvements, RSA cryptography is slower than the commonly used symmetric-key encryption algorithms such as DES (typically, in software, DES is 100 times faster than RSA and in hardware can be between 1,000 to 10,000 times faster depending on the implementation). In practice, RSA encryption is most commonly used for the transport of symmetric-key encryption algorithm keys and for the encryption of small data items.

RSA Laboratories' recommended standards can be found in [23]. RSA Laboratories recommended in 1999 that the current industry standards for the RSA algorithm, such as the ANSI X9.31 [24] banking standard for RSA signatures, require a minimum of 1024 bits for an additional level of security.

When entity *B* wants to send a signed message *M* to entity *A*, first *B* computes his signature for message M with his private decryption key (D_B):

$$S = D_B(M) \tag{2.21}$$

B then encrypts the signed message using *A*'s public key:

$$C = E_A(S) \tag{2.22}$$

A can decrypt the encrypted signed message with his private decryption key (D_A):

$$S = D_A(C) \tag{2.23}$$

$$S = D_A(E_A(S)) \tag{2.24}$$

A can recover the message using *B*'s public encryption key E_B:

$$M = E_B(S) \tag{2.25}$$

$$M = E_B(D_B(M)) \tag{2.26}$$

Hence, RSA signature scheme recovers the message from the signature, the sender does not need to send the encrypted message separately with the signature.

2.4.3 RSA Problem (RSAP)

The RSA Problem is defined as follows, given a positive integer $n = pq$, where p and q are two distinct odd primes; a positive integer e such that gcd(e; (p-1)(q-1)) = 1; Find the plaintext integer m such that $m^e \equiv c \pmod{n}$. Therefore, one has to find the e^{th} roots modulo a composite integer n. There is exactly one $m \in (0, 1, \ldots, n\text{-}1)$ for each integer $c \in (0, 1, \ldots, n\text{-}1)$. Rivest and Kaliski [25] provides a good insight into the RSA Problem.

RSA assumption is that the security of RSA depends on large integer factorisation problem: RSA Problem becomes difficult to solve when the modulus n is sufficiently large and both p and q are two large random and distinct prime numbers, therefore both the plaintext m and ciphertext c is a random number between 0 to n-1. It is important for plaintext m to be random and be over a wide range of [0, n-1], otherwise an adversary can compute m by trying all possible values for m.

Factoring is believed to be a mathematically difficult problem, i.e. *NP* complete, it has not yet been mathematically proven and an efficient factoring algorithm remains to be discovered, however, this is widely believed to be unlikely.

It has been said that the RSA problem is closely related to factoring, Boneh and Venkatesan show the RSA problem may not be equivalent to integer factorisation when the public exponent is small [26], an example is given where public exponent e = 3. Another RSA problem consideration is that, it is not necessarily true that a large number is more difficult to factor than a small number. However, it is known that a number with large prime factors is more difficult to factor than a number with small prime factors; hence, large modulus should be used for an RSA cryptosystem. There are other rules in choosing the modulus in order to preserve security of RSA, as mentioned before, the two primes, p and q, which form the modulus should be of roughly equal length. In addition, one should be aware that if the two primes are too close together, it increases the ease of determining the values by the adversary, since:

If $p \approx q$, let mean of p and $q = m = [(p+q)/2]$; then p can be determined as:

$$p = m \pm \sqrt{(m^2 - n)}; \tag{2.27}$$

Where $n = pq$.

However, the probability of this happening in reality is low. Another concern over increasing key size is that the overall RSA algorithm operations will also take longer. For example, doubling the length of the modulus will on average increase the time required for encryption and signature verification, (which made use of the public key) by a factor of four, and increase the time taken by the decryption and signature operation, (which make use of the private key) by a factor of eight. The reason why public key operations are affected less than private key operations is that the public exponent can remain fixed while the modulus is increased, whereas the length of the private exponent increases proportionally. One should choose a modulus (key) length with the following considerations:

1. The value of the protected data and the length of time it needs to be protected;
2. How powerful are the threats.

2.4.4 Security of RSA

In the previous section, the security of RSA cryptosystems relating to RSA problem and factoring was discussed. In this section, more security issues and attacks on RSA cryptography will be reviewed. Boneh provides a good general insight into security of RSA encryption [27].

1.- Attack relating to RSA factoring problem / Chosen Cipher Attack

As mentioned previously, public key cryptography is susceptible to chosen cipher attacks, RSA is also prone to this type of attack. Some characteristics of RSA can be exploited to perform chosen cipher attack, such as multiplicative property of RSA. This is described in [28]. More chosen cipher attack on RSA are described in [29] and [30].

An effective method to defeat such an attack is known as Optimal Asymmetric Encryption Padding (OAEP) [31]. The objective of OAEP is to mask the plaintext message M with the hash G of a random number r and this string of masked data is concatenated with the XOR of hash H of the mask data ($M \oplus G(r)$) with random string r as shown follows:

$$[M \oplus G(r)] \| [r \oplus H(M \oplus G(r))] \qquad (2.28)$$

|| denotes concatenation. Different variations of OAEP can be found in [32] and [33].

2. **Small Encryption Exponent *e***

The advantage of using small encryption exponent e, such as $e = 3$ is that faster public key encryption and faster public key signature verification can be gained. However, it has been shown by Hastad [34] that having small encryption key could be insecure when the same plaintext is sent to many different recipients. Each recipient has their own public moduli, since these moduli are most likely pair-wise relatively prime, therefore an attacker could quite easily compute the plain text using Gauss's algorithm. In such cases, small encryption exponent should be avoided.

3. **Forward Search Attack**

As mentioned previously, the range of message must be large and unpredictable, otherwise an adversary can decrypt a ciphertext by encrypting all possible plaintext. One method to avoid this attack is to append pseudorandom bit-string to the pre-encrypted plaintext message.

4. **Small Decryption Exponent *d***

To speed up RSA signature generation or decryption time, one may choose to use small decryption exponent *d*, however, Boneh and Durfee [35] shows that when the private key used in the RSA public key cryptosystem is less than $N^{0.25}$, the system is insecure. Wiener [36] also proposed an attack on RSA when small decryption exponent *d* is used. To avoid this attack, the decryption exponent *d* should be roughly the same size as *n*.

5. Multiplicative Properties

Let m_1 and m_2 be two plaintext messages and let c_1 and c_2 be their respective RSA encryption:

$$(m_1 m_2)^e \equiv m_1^e m_2^e \equiv c_1 c_2 (\bmod n) \qquad (2.29)$$

This means that the plaintext $m = m_1 m_2 \bmod n$ and the corresponding ciphertext is $c = c_1 c_2 \bmod n$. Due to this property of RSA, adaptive chosen cipher attack can be performed, and the adversary can retrieve the plaintext. This can be avoided by applying pre-defined structure constraints on the plaintext. All ciphertext decrypted to a message which does not possess the same pre-defined structure will be rejected as fraudulent.

6. Common Modulus Attack

RSA system where the users within an organization would share the public modulus is susceptible to this type of attack. For example, the administration would choose the public modulus *n*, two users would then have their encryption and decryption key generated $((\mathrm{e}_1, \mathrm{n}), \mathrm{d}_1))$ and $((\mathrm{e}_2, \mathrm{n}), \mathrm{d}_2))$ from the same modulus. The eavesdropper can recover the plaintext by doing the following:

$$C_1 = M^{e1} \bmod n \tag{2.30}$$

$$C_2 = M^{e2} \bmod n \tag{2.31}$$

$$(e_1)a + (e_2)b = 1 \qquad if\ \gcd(e_1, e_2) = 1 \tag{2.32}$$

$$M = C_1^{\,a} + C_2^{\,b} \bmod n \tag{2.33}$$

To avoid this attack each entity should choose its own RSA modulus *n*. Since any knowledge of encryption and decryption key pair allow for the factorisation of the modulus *n*.

7. Cycling Attack

Given $c = m^e \bmod n$, there exists a *k* such that $c^{e^k} \bmod n = c$, so $c^{e^{k-1}} \bmod n = m$. $c^{e^k} \bmod n$ is then computed until *c* is obtained and the previous power is the message. However, this attack is considered non-threatening to the security of RSA since factoring *n* is assumed to be intractable.

8. Message Concealing

A message is unconcealed when it encrypts back to itself, i.e. $m^e \equiv m(\bmod\ n)$, however, this cannot be avoided since there will always be some messages which are unconcealed, such as, when $m = 0$, $m = 1$ and $m = n$-1. Even though this might be the case, it does in actuality pose a threat since the proportion of the unconcealed message is small as *p* and *q* are randomly chosen prime numbers, and *e* is also chose at random.

2.4.5 ElGamal

The ElGamal system [17] is a public-key cryptosystem, which unlike RSA algorithm is based on the discrete logarithm problem, where the security of RSA is based on integer factorisation. It is commonly used for both encryption and signature. The ElGamal encryption algorithm is similar in nature to the Diffie-Hellman key agreement protocol, the system contains a prime integer p and an integer called generator g, whose power modulo p generates a large number of elements. Each entity creates a public key and a corresponding private key. Entity A creates the key pair and sends public key information to entity B. B will then encrypt the message with A's public key. A decrypts the message with his/her own private key. All entities can choose to use the same prime p and generator g. If common parameters are chosen, they do not need to be published as part of a public key. The advantage of having a common parameter is that the computation can be sped up by using precomputations. The disadvantage is that a larger moduli p may be needed in case of security being compromised.

The ElGamal system is based on discrete logarithm problems like the Diffie-Hellman system. For any cryptographic system that is based on discrete logarithm problems, the chosen large prime p must be chosen such that (p-1) has at least one large prime factor, otherwise the security of the system will be compromised. Discrete logarithm problems apply to mathematical structures called groups, where a group consists of a set G which could be finite or infinite, together with a binary operation called group multiplication. This group multiplication is defined as:

$$* G \times G \rightarrow G \tag{2.34}$$

This means that the product of $a*b$ is $\in G$ for any two elements a and $b \in G$. A group consists of the following properties:

1. Associatively: The operation * is associative, i.e. $a*(b*c) = (a*b)*c$ for any $a, b, c, \in G$.

2. Identity element: For each element $a \in G$, there is an identity element where $a*e = e*a = a$.

3. Inverse element: For each element $a \in G$, there is an inverse element, such that $a*b = b*a = e$ which is the identity element where $b \in G$.

The group G is said to be closed for all $a, b \in G$, $a^* b \in G$, also the group G is said to be Abelian (or commutative) if $a^*b = b^*a$ for all $a, b \in G$.

For $g \in G$ and a number n, g^n means that g is multiplied itself n times, e.g. $g^3 = g^*g^*g$ and the discrete logarithm problem is defined as follows:

Let $g \in G$ and $h \in G$, find the value x such that:

$$g^x = h \tag{2.35}$$

The basic ElGamal Encryption scheme is defined as follows:

1. Key generation for ElGamal public key encryption:

    a. Generate a large random prime number p and a generator g of the multiplicative group Z_p^* of the integer modulo p.

    b. Select a random integer a, where a is $1 \le a \le p$-2, then compute g^a mod p.

    c. Public key: (p, g, g^a); Private key: a.

2. ElGamal public encryption:

    a. Obtain recipient's authentic public key (p, g, g^a).

    b. Represent the message as an integer m in the range {0, 1, ..., p-1}.

    c. Select a random integer k, where k is $1 \le k \le p$-2,

    d. Compute the γ and δ:

$$\gamma = g^k \bmod p \tag{2.36}$$

$$\delta = m \cdot (g^a)^k \bmod p \tag{2.37}$$

e. Ciphertext $c = (\gamma, \delta)$

3. ElGamal decryption:

a. Compute the following:

$$\gamma^{p-1-a} \bmod p \tag{2.38}$$

Where a is the private key of the recipient and $\gamma^{p-1-a} = \gamma^{-a} = g^{ak}$.

b. Recover the message m by computing the following:

$$(\gamma^{-a}) \cdot \delta \bmod p \tag{2.39}$$

Note that the ElGamal Encryption scheme requires two modular exponentiations, and the ciphertext is double the size of the message. ElGamal algorithm is slower compared with the RSA algorithm, particularly for signing. The randomness required by this encryption scheme reduces the effectiveness of a chosen-cipher attack. As mentioned before, the security of the ElGamal system is based on discrete logarithm problem in Z_p^*, however, it has not yet been proven that this is equivalent to a discrete logarithm problem in Z_p^*, on the other hand, the ElGamal system is equivalent to the Diffie-Hellman problem. In order to ensure the security of the system, the randomness of random integer k must be ensured, it is very important that a different random integer k is used for encrypting the different messages. Apart from the multiplicative group Z_p^*, the ElGamal system is also suitable for the following groups:

1. The multiplicative group F_{2m}^* of the finite field F_{2m} of characteristic two.

2. The multiplicative group F_q^* of the finite field F_q, where $q = p^m$ and p is a prime.

3. The group of units Z_n^*, where n is a composite integer.

4. The class of group of an imaginary quadratic number field.

5. The group of points on an elliptic curve over a finite field.

6. The jacobian of a hyperelliptic curve defined over a finite field.

The ElGamal Digital Signature scheme is as follows:

The key generation process is the same as ElGamal encryption as shown before:

- Public key: (p, g, g^a)
- Private key: a.

1. Signature generation:

   a. Choose random secret integer k, where k is $1 \leq k \leq (p\text{-}2)$ and gcd $(k, p-1) = 1$.

   b. Compute the followings:

   $$r = g^k \bmod p \qquad (2.40)$$

   Solve for s in the signing equation:

   $$m = ar + ks \bmod (p-1) \qquad (2.41)$$

   $$\therefore s = k^{-1}(m - ar) \bmod (p-1) \qquad (2.42)$$

   c. The digital signature is (r, s) and the signed message is $(m, (r, s))$

2. Signature verification:

   a. Verify that $1 \leq r \leq p\text{-}1$; if not, reject.

   b. Accept the signature if and only if

$$(g^a)^r r^s \bmod p = g^m \bmod p \tag{2.43}$$

As mentioned before, DSS is NIST's modification of ElGamal Signature Scheme, however, DSS is only useful for signing, and it is not good for encryption. The differences between DSS and ElGamal Signature scheme is as follows:

- Use + instead of – in signature verification

- Introduce prime q which is a 160-bit prime factor of (p-1) and hence signature (r, s) has been changed to the following:

$$r = (g^k \bmod p) \bmod q \tag{2.44}$$

$$s = k^{-1}(h(m) + ar) \bmod q \tag{2.45}$$

- p is a 512-bit (revised up to 1024-bit) prime such that q divides (p-1)

- q is 160-bit prime factor of (p-1)

- h is a 160 bit one-way hash function

- 512-bit Public key

- 160-bit private key a

2.5 Comparisons: Symmetric Key Cryptography vs. Public key Cryptography

Table 2.1 – Advantages and disadvantages of Symmetric Key Algorithm

Advantages	• *High rates of data throughput, especially for hardware implementation and less processor-intense*
	• *Relatively short key needed*
	• *Can be used as primitives to construct other cryptographic mechanism, e.g., hash functions and digital signature scheme*
	• *Can be transformed into strong product ciphers*
	• *Long history*
Disadvantages	• *Key distribution problem – key must be kept secret, particularly in large communication network*
	• *Key management problem in large network – need unconditionally trusted TTP*
	• *Keys have to be changed as frequently as each session in a two party communication*
	• *Symmetric key signature scheme requires large key for verification or need trusted TTP*

Table 2.2 – Advantages and disadvantages of Public Key Algorithm

Advantages	• *Only need to keep private key secret, however, the authenticity of the public key must be ensured through other means*
	• *For key management, only a functionally trusted third party (TTP) is needed (Offline manner), instead of a unconditionally trusted TTP that is required by TTP (Real-time)*
	• *Depending on the type of communication, the key pair can be reused many times*

	• *Relatively efficient digital signature scheme*
	• *Suitable in a large network communication, considerably smaller keys required than in case of symmetric key system*
Disadvantages	• *Throughputs rates for most encryption methods are several magnitudes slower than that of common symmetric key system*
	• *Need large key size compare with symmetric key (by a factor of 10 or more) to minimise the chance of short-cut attacks (e.g. factoring); the most effective attack on symmetric key schemes is exhaustive key search.*
	• *The security of public key systems are based on a presumed "hard" problem of number theory, however, it has not been proven to be secure*
	• *Relatively short history, discovered in mid 1970s*

Table 2.1 and Table 2.2 show the advantages and disadvantages of symmetric key and public key system respectively. In summary, both symmetric key scheme and public key system have different complementary advantages. In order to make use of the strength of both systems, a so called hybrid encryption can be used. Since symmetric key encryption is more efficient and public key cryptography has the benefit of having reusable public and private key pairs, the best way to make use of both schemes is to use a symmetric key scheme for encryption of the message and a public key system for key establishment and management. Furthermore, a public key can also be used for signing digital signature and encrypting session key. Because of the low throughput rates of public key encryption compared with symmetric key encryption, public key encryption is only suitable for encrypting small data, whereas symmetric key encryption systems can be used for encrypting larger sized data. By employing the two schemes as mentioned earlier, the cryptographic system can achieve encryption efficiency provided by symmetric key encryption and attain the non-repudiation and authentication objectives with the public key system digital signature and secure key exchange provided by public key management. One example of a cryptographic system that makes use of such arrangement is called

PGP2 (Pretty Good Privacy 2) [37]. This was created by P. Zimmermans in 1991, this design makes use of both RSA and IDEAL.

When comparing symmetric key with public key systems, apart from their functionality, another important concern is their key sizes and equivalent security level. Traditionally, the strength of security of an algorithm given the key size is described as the amount of time it takes to exhaust all possible keys for a symmetric algorithm. NIST recommendation for the key size used for RSA system should be 1024 bit or higher for long-term security [38].

Table 2.3 can be found in [38], it provides the equivalent strength of different algorithms with the recommended key size, such as 1024-bit RSA should have equivalent security to 80-bit symmetric key. The final column of the table shows the equivalent security strength of elliptic curve cryptography. Elliptic curve cryptography was first proposed by Victor Miller and Neal Koblitz independently in the mid 1980s. Elliptic curve cryptography is an approach to public key system making use of the mathematics of elliptic curves. When defining an elliptic curve system, a curve and a base point are required. Elliptic curve cryptography will be discussed later on in this chapter.

Table 2.3 - Equivalent Strength

Bits of Security	Symmetric key Algorithm	Hash Algorithm	Discrete Logs (DSA, DH, MQV)	RSA	Elliptic Curves
80		SHA-1	$L = 1024$ $N = 160$	$k = 1024$	$f = 160$
112	TDES		$L = 2048$ $N = 224$	$k = 2048$	$f = 224$
128	AES-128	SHA-256	$L = 3072$ $N = 256$	$k = 3072$	$f = 256$
192	AES-192	SHA-384	$L = 7680$ $N = 384$	$k = 7680$	$f = 384$
256	AES-256	SHA-512	$L = 15360$ $N = 512$	$k = 15360$	$f = 512$

The following explains each column of Table 2.3 [38]:

- Column 1 indicates the number of bits of security provided by the algorithms and key sizes in a particular row.

- Column 2 provides the symmetric key algorithms that provide the indicated level of security, where TDES is approved in FIPS46-3 [5] and specified in ANSI X9.52 [39], and AES is specified in FIPS197 [11]. Note: it is assumed in the table that TDES is using three distinct keys.

- Column 3 provides the equivalent hash algorithms that are specified in FIPS180-2 [40] for the given level of security.

- Column 4 indicates the size of the parameters associated with the standards that use discrete logs (DSA as defined in FIPS186-3 [22] for digital signatures, and Diffie-Hellman (DH) and MQV key agreement as defined in ANSI X9.42 [41] and SP 800-56 [42]), where L is the size of the modulus p, and N is the size of q. The value of L is considered to be the key size.

- Column 5 defines the value for k (the size of the modulus n) for the RSA algorithm specified in ANSIX9.31 [24] and PKCS1 [43] and adopted in FIPS186-3 [22] for digital signatures, and specified in ANSIX9.44 [44] and adopted in SP 800-56 [42] for key establishment. The value of k is commonly considered to be the key size.

- Column 6 defines the value of f (the order of the base point G of the selected elliptic curve) for the elliptic curve algorithms specified for digital signatures in ANSIX9.62 [45] and adopted in FIPS186-3 [22], and for key establishment as specified in ANSIX9.63 [46] and adopted in SP 800-56 [39] The value of f is commonly considered to be the key size.

As discussed earlier, due to different strengths of different schemes, a combination of different algorithms can be used to achieve optimized cryptographic results. Table 2.4 [38] provides recommendations that could be used to select the appropriate set of algorithms with their appropriate key sizes. [38] suggests a minimum of 80 bits symmetric algorithm equivalent are adequate for most applications until year 2015, for longer term security, minimum of 112 bits is recommended.

Table 2.4 - Recommended algorithms and minimum key sizes

Years	Symmetric key algorithms (Encryption & Mac)	Hash Algorithm	HMAC	DSA	RSA	Elliptic Cueves
Present – 2015	TDES AES-128 AES-192 AES-256	SHA-1 SHA-256 SHA-384 SHA-512	SHA-1 (≥80 bit key) SHA-256 (≥128 bit key) SHA-384 (≥192 bit key)	Min: $L = 1024$ $N = 160$	Min: $k = 1024$	Min: $f = 160$
2016 and beyond	TDES AES-128 AES-192 AES-256	SHA-256 SHA-384 SHA-512	SHA-256 (≥128 bit key) SHA-384 (≥192 bit key) SHA-512 (≥256 bit	Min: $L = 2048$ $N = 224$	Min: $k = 2048$	Min: $f = 224$

The followings explain each column of Table 2.4 [38]:

- Column 1 indicates the years during which the algorithms specified in subsequent columns are appropriate for use.

- Column 2 identifies appropriate symmetric key algorithms and key sizes: the Triple DES algorithm (TDES) is specified in FIPS46-3 [5], the AES algorithm is specified in FIPS197 [11], and the computation of Message Authentication Codes (MACs) using block ciphers is specified in SP800-38b [47].

- Column 3 specifies the hash sizes to be used for most hash applications (e.g., digital signatures). Hash algorithms are specified in FIPS180-2 [40].

- Column 4 specifies the hash algorithm and minimum key size to be used for keyed-hash (HMAC) computations. HMAC is specified in FIPS198 [48].

- Column 5 indicates the minimum size of the parameters associated with DSA as defined in FIPS186-3 [22].

- Column 6 defines the minimum size of the modulus for the RSA algorithm specified in ANSIX9.31 [24] and PKCS1 [43] and adopted in FIPS186-3 [22] for digital signatures, and specified in ANSIX9.44 [44] and adopted in SP 800-56 [40] for key establishment.

- Column 7 defines the minimum size of the base point for the elliptic curve algorithms specified for digital signatures in ANSIX9.62 [45] and adopted in FIPS186-3 [22], and for key establishment as specified in ANSIX9.63 [46] and adopted in SP 800-56 [49].

Even though it has been suggested that 1024-bit RSA should be sufficient to last for another 10 years for general data (*see Table 2.4*) [38], there has been concern that this may not be the case. More and more research is being carried out aiming at improving the technical aspect of integer factorization problems such as the well-known method Number Field Sieve. This research could threaten the strength of security of 1024-bit RSA, such as [49] suggested implementation techniques of Number Field Sieve to reduce the amount of memory required to break very large RSA keys. Shamir and Tromer presented a paper on a custom-built hardware device for performing the sieving step of the Number Field Sieve algorithm in 2003 called TWIRL [50]. The paper gave a hypothetical estimation that all the sieving required for factoring 1024-bit integers can be completed within 1 year by a device that costs $10M to manufacture plus a one-time cost of $20M for all the pre-device cost such as design, simulation, mask creation etc. Based on analysis of all this recent research, RSA Security© produced a technical note [51] which provides the following recommendation for key sizes:

Table 2.5 - Recommended minimum symmetric security levels and RSA key sizes based on protection lifetime

Protect Life of Data	Present - 2010	Present - 2030	Present – 2031 and Beyond
Minimum symmetric security level	80 bits	112 bits	128 bits
Minimum RSA key size	1024 bits	2048 bits	3072 bits

[51] recommended that 112-bit security is possibly higher than needed for present time, but it should be convenient for implementation since triple-DES is readily available and 2048-bit RSA key size is also convenient as it is already recommended for use in root keys. As an interim measure, a minimum 1536-bit RSA signature is reasonable, as recommended by New European Schemes for Signatures, Integrity and Encryption (NESSIE) [52], however, due to the complexity of the upgrading process, RSA Security© [51] advised that 2048 bits is a better goal.

Increasing the key length provides increase in security against Brute-force attack, however, It reduces the performance of the system because the number of cycles of computation involved also increases (providing the same size hardware is used). In the next section, elliptic curve cryptography will be introduced. This type of cryptographic system requires a smaller key but the security is not compromised. This will be discussed in further details in the next section.

2.6 Elliptic Curve Cryptography (ECC)

In the previous section, two main families of public key algorithms were introduced; integer factorisation schemes, e.g. RSA and discrete logarithm schemes, e.g. Diffie-Hellman. There exists another form of cryptographic scheme and it is called Elliptic Curve Cryptography (ECC). In this section, ECC encryption and other applications will be explained, however the details of elliptic curve algebraic theory and finite field arithmetic will be explained in Chapter 3.

Elliptic Curve Cryptography was first proposed by Miller [53] and Koblitz [54] in 1985. They independently proposed the idea of using a group of points on an elliptic curve to perform necessary cryptographic operations. Like other public key systems, ECC relies on difficult mathematical problems. Some common ECC cryptographic schemes are analogous to other public key schemes are:

- Elliptic Curve Discrete Logarithm Problem (ECDLP)
- Elliptic Curve Diffie-Hellman (ECDH)
- Elliptic Curve Digital Signature Algorithm (ECDSA)

Elliptic curve schemes that analogue to RSA can also be implemented; however, it provides no realistic benefits compare with an RSA system because of the complex calculation involved in elliptic curve arithmetic.

2.6.1 Elliptic Curve Discrete Logarithm Problem (ECDLP)

As mentioned previously, the security of cryptographic systems rely on hard mathematical problems that are computationally infeasible to solve. The foundation of ECC lies upon elliptic curve discrete logarithm problem (ECDLP). ECDLP is based on the intractability of scalar multiplication products (more details on scalar multiplication can be found in Chapter 3).

ECDLP can be defined as follows:

> Given an elliptic curve group E(k), where k is a finite field, points Q and P are points in the group, find the discrete logarithm k of Q to the base P, such that $P = Q$. k should be large enough so that it is computational infeasible to exhaustively search for the discrete logarithm. [53] [54].

2.6.2 Elliptic Curve Diffie-Hellman (ECDH)

Elliptic Curve Diffie-Hellman (ECDH) [55] is analogue of Diffie-Hellman key exchange algorithm; therefore, ECDH is for key exchange prior to the use of a private key cryptosystem. In order to establish a common key before the encryption process, both entity A and B follow the following steps:

1. Fix a finite field Fq, an elliptic curve E defined over it and a base point $B \in E$.

2. Choose a random integer $a \in \mathrm{F}q$ for entity A and $a \in \mathrm{F}q$ for entity B as secret key.

3. Calculate public key:

   a. Entity A:

   $$aB \in E \tag{2.46}$$

   b. Entity B:

   $$bB \in E \tag{2.47}$$

4. The common key is:

$$P = abB \in E \tag{2.48}$$

To perform elliptic curve algorithm analogous to that of ElGamal scheme, entity A and B perform step 1-3 as shown above. If entity A wants to send a message P to entity B, A needs to perform the following steps:

1. Choose a random integer k.

2. Compute and send $(kB, P+k(bB))$ to B.

To decrypt the message, B has to multiply the first point of the point pair by his secret key b: $b(kB)$, then subtract this from the second point of the point pair:

$$P = (P+k(bB)) - b(kB) \tag{2.49}$$

2.6.3 Elliptic Curve Digital Signature Algorithm (ECDSA)

Elliptic Curve Digital Signature Algorithm is equivalent to the digital signature schemes and is approved by NIST under FIPS 186-2 [22]. ECDSA is described in ANSI X9.62 [45]. The process of ECDSA, where entity *A* is to send a digitally signed message *M* to entity *B* and entity *B* is to verify the message is from entity *A*, can be described by the following steps:

- Entity *A*:
    - Setting up the elliptic curve
        - Choose a finite field F*q*
        - Choose an elliptic curve *E* over the field
        - Set a base point *G* with order *n*
        - Private key: *d*
        - Public key: *Q*
    - Signature Generation
        - Choose a random number *k* where *k* is $1 \leq k \leq n$-1
        - Compute:

      - $kG = (x_1, y_1)$ (2.50)
      - $r = x_1 \bmod n$ (2.51)
      - *if r = 0, re-choose random number k*
    - Compute $k^{-1} \bmod n$
    - Compute e = SHA-1 (M)
    - Compute $s = k^{-1} (e + dr) \bmod n$; *if s = 0, re-choose random number k*
    - Signature for message M is (r, s)

- Entity B:
  - Signature Verification
    - Verify r, s are integers in the interval $[1, n\text{-}1]$
    - Compute e = SHA-1 (M)
    - Compute $w = s^{-1} \bmod n$
    - Compute:
      - $\mathrm{u1} = ew \bmod n$ (2.52)
      - $\mathrm{u2} = rw \bmod n$ (2.53)
    - Compute $X = u_1G + u_2Q$, where $X = (x_1, y_1)$

- *if* $X = O$, reject signature

- Else, compute $v = x_1 \bmod n$

- If v = r, accept signature

2.6.4 ECC vs. RSA

Table 2.3 shows that 160-bit ECC is equivalent to 1024-bit RSA in terms of security, which is equivalent to 80-bit symmetric key. Therefore, ECC could provide similar level of security compared with RSA but require smaller key size, hence, smaller register size and also faster processing speed. Secondly, in order to match the security of 112-bit symmetric key, ECC only needs to increase its key length to 224-bit, which is an increase of about 1.3 times, whereas RSA system needs to double its key length to match the level of security. Another benefit of having shorter key length is that it enhances the resistance to differential power analysis [1], this will be explained further in Chapter 3. Structurally, ECC is very different to RSA, this can be shown by Table 2.6 taken from [1] [56]. Table 2.6 shows that setting up system parameters for ECC is more complex compared to RSA, however, for public and private key generation, ECC is relatively easy to generate. An ECC private key is a random number k, whereas RSA private key is $d = e^{-1} \bmod ((p-1)(q-1))$. ECC public key is kG, which is a simple calculation, whereas RSA public key pair is (n, e), where e is just a random integer, $n = pq$, where pq are two large prime numbers.

Table 2.6 - System requirements for elliptic curve cryptosystems and RSA

	ECDSA and ECES	*RSA*
System parameters	• The field *F* • Two field elements that represent the curve • The generator *G* on the curve • The order of *G*	• None
Public key	• Point $P = kG$ on the elliptic curve	• Modulus *n* • Exponent *e*
Private key	• An integer *k* where $0 < k < q$	• Exponent *d* *Or* • Corresponding CRT information

Table 2.7 shows a very basic comparison in terms of storage requirements in bits of 1024-bit RSA (with public exponent 2^{16}+1) and with ECC over GF(q) where *q* is 160-bit in length and the field is either of characteristic 2 or of odd characteristic [56]. In this comparison, ECC has a lower storage requirement compare with RSA based on the system parameters and keys needed.

Table 2.7 - The storage requirements in bits when making a naive comparison between an elliptic curve cryptosystem over GF(*q*) where *q* is 160 bits in length and RSA with a 1024-bit modulus

	ECDSA and ECES over GF(q)	*RSA 1024-bit n and e = 2^{16}+1*
System parameters	(4 x 160) +1 = 641	0
Public key	160 + 1= 161	1024 + 17 + 1041
Private key	160 (801 with system parameters)	2048 (or 2560 with *CRT* information)

Another very important advantage of ECC is that there is not yet a known sub-exponential algorithm for ECDLP; this implies ECDLP should be more secure than conventional discrete logarithm cryptosystems. Current algorithms that are used for solving conventional discrete logarithm systems cannot be applied to solving ECDLP, since these algorithms are of sub-exponential time.

The disadvantage of ECC is that care needs to be taken when setting up ECC system parameters, which includes selecting the appropriate curves. There are curves that are known to be susceptible to attacks and compromise on the security of the system, such as supersingular elliptic curves [57] and Koblitz curves [58]. Another disadvantage of ECC is that it is relatively new compared with RSA, and therefore it is less well established and studied.

In conclusion, because ECC requires small key size for a similar level of security compared with other public key systems, ECC is particularly useful in computationally and power constrained environments, e.g. wireless computation and smart cards. ECC is also very useful for areas that require heavy workload, such as secure server networks. It also saves bandwidth for communications overhead.

3 Finite Field Arithmetic in Hardware and Literature Review

The first part of this chapter will be dedicated to exploring the operations of elliptic curve and finite field arithmetic involved, understanding the nature of elliptic curve is essential to the design of the unified field multiplier for GF(p) and GF(2^n) (Note that GF(p) and GF(2^n) can also be expressed as F_p and F_2n respectively). The second part of this chapter will explore attacks that need to be considered, which would severely undermine the security of the cryptographic system. Cryptanalysis is an important part of the study of cryptography. By understanding more about how a cryptosystem could be attacked, techniques against attacks can be employed or certain well-known weakness, such as weak curve, can be avoided. Finally, previously designed unified field multipliers will be reviewed.

3.1 What is Elliptic Curve?

The study of elliptic curve mathematics has been going on for many years, it was only in 1985 that the use of elliptic curve on cryptography was first proposed.

There are two different forms of elliptic curve:

1. Montgomery Form:

$$E^M : BY^2 = X^3 + AX^2 + X \qquad (3.1)$$

2. Weierstrass Form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad (3.2)$$

Weierstrass Form is the most standard form of elliptic curves. The definition presented above is so called the "long Weierstrass form", which is valid for any field. However, only finite fields are used in cryptography, therefore simpler equations are generated and an example of a short Weierstrass form curve can be represented as follows:

$$E : y^2 = x^3 + ax + b \tag{3.3}$$

Value x and y in shown equation 3.3 are variables and value a and b are constant values from the chosen field. Montgomery [59] introduced this non-standard form in 1987. Okeya *et. al.* [60] provided evidence that elliptic curve cryptosystem based on Montgomery Form are immune to timing attacks [61] [62], which is a form of side channel attacks based on timing information retrieved by the attacker. This will be investigated later on in section 3.4.5. It is possible to transfer Weierstrass form to Montgomery form, providing the following criteria are met [60]:

1. The equation $x^3+ax+b = 0$ has at least one root in finite field F_p, where p is a prime and $p \geq 5$.

2. The number $3\alpha^2 + \alpha$ is quadratic residue in F_p, where α is a root of the equation $x^3+ax+b = 0$ in F_p.

Reference [63] provides more proof on why not every elliptic curve over a prime field can be transformed into Montgomery form over the same prime field. The reason for that is because Montgomery-form elliptic curves with co-factor 4 over $F_{p'}$ are more numerous than Weierstrass-form elliptic curves with co-factor 1 over F_p (Weierstrass-form elliptic curve defined over the field F_p with co-factor 1, is security equivalent to a Montgomery-form elliptic curve with co-factor 4 defined over the field $F_{p'}$ which is larger than F_p by two bits). Please refer to [63] for details.

Any Montgomery Form elliptic curve can be transformed into Weierstrass Form. More details on these transformations can be found in [60]. Okeya [64] compares Montgomery form with Weierstrass form, and shows that the scalar multiplication on a Montgomery form elliptic curve is faster than that on a Weierstrass form elliptic curve if the size is smaller than 391 bits.

The self-evaluation reports published by Hitachi Ltd [63] provide more insight into the comparisons on application of Montgomery form and Weierstrass form. This report showed that even though Montgomery form elliptic curves are restricted curves, they have enough generality to be used for cryptosystems securely. It concluded that the security of cryptosystems using Montgomery-form elliptic curves is equivalent to the security of cryptosystem using Weierstrass-form elliptic curve providing a suitable size of definition field is chosen.

The points of the curve could be represented in two forms [65]:

1. Affine Coordinates
2. Projective Coordinates

The Weierstrass equation in the projective plane is represented as follow:

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \qquad (3.4)$$

This is a homogeneous equation of degree 3. The definition of a homogeneous polynomial is that every term in the polynomial has the same total degree, which is 3 in this case. It is possible to convert the point representation between affine and projective coordinates.

Given a point ***P***$(x, y) \in$ ***E*** (Fq) in affine coordinates, its projective coordinates equivalent is ***P'***$(X, Y, Z) \in$ ***E*** $(\boldsymbol{Fq})$, where $x = X, y = Y$ and $Z = 1$, therefore:

$$P(x,y)=P'(X,Y,Z) \qquad (3.5)$$

$$P(x,y) = P'(\frac{X}{Z},\frac{Y}{Z},1) \qquad (3.6)$$

$$P(x,y) = P'(\frac{X}{Z},\frac{Y}{Z}) \qquad (3.7)$$

The computation of the curve operation for different coordination systems is different. For a non-supersingular curve over $K = F_{2m}$, the number of field operations required to perform point addition and point multiplication in affine and projective coordinates is shown in Table 3.1 [66]. ESUM denotes elliptic curve field addition and EDBL denotes elliptic curve field doubling.

Table 3.1 - Number of field multiplications and inversions for affine and projective point addition and doubling

Operation	Affine		Projective	
	ESUM	EDBL	ESUM	EDBL
Field Multiplication	2	3	13	7
Field inversion	1	1	0	0

Projective coordinates does not involve field inversion calculation, however more field multiplication is required. Therefore in the case where inversion is much slower than multiplication, calculations in projective coordinates should be more efficient than that in affine coordinates.

It is widely accepted that the inversion calculation is generally less efficient than other necessary ECC operations, such as multiplication. Reference [67] shows timings in μs required by software implementation of different binary field operations carried out in field GF $_2{}^{163}$, GF $_2{}^{233}$ and GF $_2{}^{283}$ on a 1000 MHz Pentium III, as shown in Table 3.2. Notice the ratio of inversion to multiplication shows that inversion operation is generally 9 - 10 times slower than multiplication. In such a cases, the number of inversions involved should be minimised, i.e. projective coordinates should be used instead of affine coordinates.

Table 3.2 - Timings (in μs) on a 1000 MHz Pentium III over Binary Field

	GF_2^{163}	GF_2^{233}	GF_2^{283}
Addition	0.032	0.039	0.041
Modular Reduction	0.081	0.094	0.145
Multiplication (incluing reduction)			
• Shift-and add	6.11	9.66	13.25
• LR (left-to-right) comb with	1.06	1.92	2.40
• Karatsuba	1.49	2.69	3.13
Squaring	0.19	0.24	0.31
Inversion	10.0	17.4	24.5
Inversion/ Multiplication ratio	9.5	9.1	10.2

The disadvantage of using projective coordinates is that they require greater temporary storage, extra registers are needed to store the points and store intermediate results when doing the addition. Therefore, in the case where memory resources are extremely constrained, affine coordinates may be a more appropriate choice. Leung [68] provides the processing time required to process point multiplication in affine and projective coordinates using the same hardware and microcode with different number of bits *n*. The results are shown in Table 3.3, and demonstrate that multiplication using affine coordinates is 10 – 23% slower than using projective coordinates.

Table 3.3 - Execution time for projective and affine coordinate implementations of elliptic curve multiplication

n	No of cycles (affine)	No of cycles (projective)	Hardware times Affine (*ms*)	Hardware times Projective (*ms*)	*Ratio P:A*
113	148581	134484	4.8	4.3	0.9
144	324717	249879	10.8	8.3	0.77
173	402926	310043	14.4	11.1	0.77

Affine and projective coordinates are the two traditional representations, however new sets of coordinates have been explored, such as Jacobian and Chudnovsky Jacobian coordinates [89], which have been researched a great deal in recent years. Reference [70] proposed a system for the use of mixed coordinates, so that the optimal set of combination could be used for calculating elliptic curve exponentiation.

3.2 Elliptic Curve Mathematical Background

Elliptic curve cryptography makes use of elliptic curve operations over finite fields. The idea of group operations was briefly explained in section 2.4.5. The addition rules for points in an abelian group are as follows [66]:
For all $P, Q \in \boldsymbol{E}$

1. $\mathcal{O} + P = P \text{ and } P + \mathcal{O} = P$

2. $-\mathcal{O} = \mathcal{O}$

3. If $Q = -P$, then $P + Q = \mathcal{O}$

4. If $P = (x_1, y_1) \neq \mathcal{O}$, then $-P = (x_1, -y_1 - a_1x_1 - a_3)$ (Note that P and $-P$ are the only points on $\boldsymbol{E}$ with x-coordinates equal to x_1)

5. If $P \neq \mathcal{O}$, $Q \neq \mathcal{O}$ and $Q \neq -P$, then let R be the third point of intersection of either the line $\overline{PQ}$ if $P \neq Q$ (see Figure 3.1), or the tangent line to the curve at P if $P = Q$ (see Figure 3.2), with the curve, where the tangent line to the curve $f(x, y) = 0$ at $P(a, b)$ is the line $\frac{\partial f}{\partial x}(P)(x-a) + \frac{\partial f}{\partial y}(P)(x-b) = 0$. Then $P + Q = -R$.

The definition of point at infinity $\mathcal{O}$ is as follows [66]:

An elliptic curve ***E*** (or an algebraic curve of genus 1) is the set of all solution in projective plane $P^2(\overline{K})$ of a smooth Weierstrass equation. There is exactly one point in ***E*** with *Z*-coordinate equal to 0, namely (0:1:0), which is called point at infinity O.

The Weierstrass equation is said to be small or non-singular if for all projective points $P = (X\text{: } Y\text{: } Z) \in P^2(\overline{K})$ satisfy the following:

$$F(X,Y,Z) = Y^2Z + a_1XYZ + a_3YZ^2 - X^3 - a_2X^2Z - a_4XZ^2 - a_6Z^3 = 0 \qquad (3.8)$$

at least one of the three partial derivatives $\frac{\partial F}{\partial X}, \frac{\partial F}{\partial Y},$ or $\frac{\partial F}{\partial Z}$ is non-zero at *P*. The Weierstrass equation is said to be singular when it possesses a singular point *P* where all three partial derivatives vanish.

A group ***G*** it said to be finite if it contains a finite set of elements, and the number of elements in the group is denoted as #***G***. A finite field or Galois field covers a finite set of points. The order of the elliptic curve ***E*** over a finite field *q* is denoted as #***E***(*q*). The two most common finite fields that are used in elliptic curve cryptography are:

1. GF(*p*) – prime field
2. GF(2^n) – binary extension field

GF(*p*) is a finite field with *p* elements where *p* is a prime number. Given an elliptic curve $E_{(a,\, b)}$(GF(*p*)), for $p \neq 2, 3$, and let $a, b \in$ GF(*p*) the inequality $4a^3+27b^2 \neq 0$, the curve can be defined as the set of points (x, y) ∈ GF(*p*) × GF(*p*), together with the point at infinity O , that satisfy the equation (3.3).

In the case where $p = 2, 3$, then the curve is said to be supersingular if and only if the *j*-invariant of *E*, *j*(*E*) = 0. A curve is said to be supersingular if *p* divides *t* where #*E*(*Fq*) = *q* + 1 – *t*, otherwise they are called non-supersingular. An example of supersingular curve is $y^2 + y = x^3 + a_4x + a_6$, they are very efficient in terms of computation, supersingular curves are not secure enough for cryptography, as motioned in section 2.6.4[57].

GF (2^n) is a finite field with 2^n elements and is represented in either polynomial or normal basis number. Given $n \geq 1$, then the non-supersingular elliptic curve can be defined as the set of solutions $(x, y) \in GF(2^n) \times GF(2^n)$, along with the point at infinity O, to the following equation:

$$y^2 + xy = x^3 + ax^2 + b \qquad (3.9)$$

GF (2^n) is particularly efficient for hardware implementation due to its binary nature. [71] provides a very good overview on the characteristic of different finite fields.

3.2.1 The order of an element

The order of an element is the smallest exponent that yields the identity element, where $j < p$:

$$a^j = 1 \bmod p \qquad (3.10)$$

3.2.2 Generator

There exists an element in all fields, when raised to a power, it gives rise to another element in the field, such that for prime number fields:

$$a = g^j \bmod p \qquad (3.11)$$

For every j between 0 and p-1, a different element in the field can be obtained. However, not every element in a field is a generator. A generator has the maximum possible order of p-1 elements.

3.2.3 Modular Arithmetic

Modular arithmetic is a very important operation in elliptic curve cryptography, since GF(p) and GF(2^n) means a number modular either a prime number p or a polynomial of degree n with binary coefficients respectively. The expression $a \equiv b(\bmod n)$ means that a and b are both in the same "congruence class" modulo n, i.e., both leave the same remainder on division by n:

$$a \bmod n \equiv r \tag{3.12}$$

$$b \bmod n \equiv r \tag{3.13}$$

Also, the difference between a and b is a multiple k of n, such that:

$$(a-b) = k \cdot n \tag{3.14}$$

The multiplicative inverse a^{-1} of a modulo n is the solution x of the congruence

$$ax \equiv 1(\bmod n) \tag{3.15}$$

where x is in the range of 1 to n-1.

3.2.4 Polynomial Basis

A polynomial is a sum of terms consisting of different powers of a variable as shown below:

$$a_{n-1}x^{n-1} + \cdots + a_1x^2 + a_1x^1 + a_0x^0 \qquad (3.16)$$

Where a_ns are the coefficients and x is the variable. For polynomial over Galios field GF$_2{}^m$, the elements of the polynomial of of degree less than m and with coefficient in F_2, therefore the coefficients $a_i \in \{0,1\}$. It is generally believed that polynomial basis is more suitable for software implementation [72] [73] [74]. The following section describes arithmetic operations in $F_2{}^m$.

1. Addition

$$a_{m-1}...a_1a_0 \oplus b_{m-1}...b_1b_0 = c_{m-1}...c_1c_0 \qquad (3.17)$$

In terms of hardware implementation, addition in $F_2{}^m$ means bitwise-XOR $c_i = a_i \oplus b_i$.

2. Subtraction

In $F_2{}^m$, addition and subtraction are equivalent, because each element is its own additive inverse, therefore:

$$(a_{m-1}...a_1a_0) + (a_{m-1}...a_1a_0) = (0...00) \qquad (3.18)$$

3. Multiplication

$$(a_{m-1}...a_1a_0) \bullet (b_{m-1}...b_1b_0) = (r_{m-1}...r_1r_0) \qquad (3.19)$$

$r_{m-1}x^{m-1} + ... + r_1 x + r_0$ is the remainder of the multiplication divided by the irreducible polynomial *f*(*x*) over F_2, where $f(x) = x^m + f_{m-1}x^{m-1} + ... + f_2 x^2 + f_1 x + f_0$. It cannot be factorised into two polynomials over F_2.

4. Exponentiation

$$(a_{m-1}...a_1 a_0)^e \tag{3.20}$$

This is equivalent to multiplying the polynomial by itself *e* times, where *e* is an integer exponent.

5. Multiplicative Inverse

Given an element generator $g \in F_2{}^m$, where $a = g^i$, the multiplicative inverse a^{-1} is:

$$a^{-1} = g^{-1} \bmod(2^m - 1) \tag{3.21}$$

3.2.5 Optimal Normal Basis

Optimal normal basis (ONB) [75] [76] is the alternative representation for elements in finite field GF$_2{}^m$. It is widely believed that optimal normal basis is more suitable for hardware implementation, because the squaring of an element is equivalent to a cyclic shift of the binary representation [77] [78] [79], because the sequence of operations for each coefficient can be parallelised easily in hardware, whereas the parallelism is difficult to implement in software.

Let $\beta \in \mathrm{GF}_2{}^m$:

$$\beta = a_n x^n + \ldots + a_1 x + a_0 \tag{3.22}$$

where $n < m$.

The normal basis of the field $\mathrm{GF}_2{}^m$ is represented as follows:

$$\{\beta^{p^{m-1}}, \ldots, \beta^{p^2}, \beta^p, \beta\} \tag{3.23}$$

For $\mathrm{GF}_2{}^m$, finite field of characteristic 2, each element A in the field can be represented as:

$$A = \sum_{i=0}^{m-1} A_i \beta^{2^i} \tag{3.24}$$

where $a_i \in F_2$ and $\beta \in \mathrm{GF}_2{}^m$.

As in the case of polynomial basis, addition is computed as bitwise-XOR and subtraction is equivalent to addition. The computation of multiplication in ONB is more complex and is described as follows:

$$A = \sum_{i=0}^{m-1} a_i \beta^{2^i} \tag{3.25}$$

$$B = \sum_{j=0}^{m-1} b_j \beta^{2^j} \tag{3.26}$$

The multiplication $C = A*B$:

$$C = A \cdot B = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \beta^{2^i} \beta^{2^j} \tag{3.27}$$

$$C = \sum_{k=0}^{m-1} c_k \beta^{2^k} \tag{3.28}$$

The sum of the basis terms:

$$\beta^{2^i} \beta^{2^j} = \sum_{k=0}^{m-1} \lambda_{ijk} \beta^{2^k} \tag{3.29}$$

The λ_{ijk} coefficient is called the lambda matrix and $\lambda_{ijk} \in \{0,1\}$, substituting the lambda matrix into the multiplication equation:

$$C_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \lambda_{ijk} \tag{3.30}$$

where $0 \le k \le n\text{-}1$

$$(\beta^{2^i} \beta^{2^j}) \cdot 2^{-l} = \sum_{k=0}^{m-1} \lambda_{i-l,j-l,k-l} \beta^{2^k} \tag{3.31}$$

$$\beta^{2^{i-l}} \beta^{2^{j-l}} = \sum_{k=0}^{m-1} \lambda_{ijk} \beta^{2^{k-l}} \tag{3.32}$$

Equate β^{2^0} to the equation above, the c_k is as follows:

$$c_k = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} \lambda_{i-k,j-k,0} a_i b_j \tag{3.33}$$

$$c_k = \sum_{i=0}^{m-1}\sum_{j-0}^{m-1} \lambda_{i,j,0} a_{1+k} b_{j+k} \tag{3.34}$$

There are two classifications of ONB and it is determined by the value of m:

1. Type I ONB

   The ONB must meet the following two criteria:

   i. m+1 must be prime.

   ii. 2 must be primitive in Z_{m+1}, where Z_{m+1} is a ring of integer modulo m+1. This means that when 2 is raised to any power in the range {0, ..., m-1} modulo (m+1), the result must be an unique integer in the range of {0, ..., m}.

2. Type II ONB

   There are two versions of Type II ONB, the ONB must meet the following two criteria:

   i. $2m$+1 must be prime.

   ii.(a) 2 is primitive in Z_{m+1}, this means that every 2^kmodulo $2m$+1, is in the range 1 to $2m$ (0 ≤ k ≤ $2m$-1). Therefore 2 is the generator for all the possible locations in the $2m$+1 field.

or

ii.(b) $2m+1 \equiv 3 \bmod 4$ and 2 generates the quadratic residues in Z_{2m+1}, this means that even if (2^kmod $2m$+1) does not generate every element in the range 1 to $2m$, $\sqrt{2k} \bmod 2m+1$could be taken so that half of the points in the field form by rule ii*a* can be generated.

This section has introduced different parameters involved in setting up an elliptic curve, it has also described their advantages and disadvantages. Since the combinations of these parameters affect the design of the architecture, one must select these parameters very carefully at the beginning of the design. For the purpose of this dual field multiplier, generality is the key to the design to provide flexibility to users. Hence the more common Weierstrass Form elliptic curve is more appropriate than Montgomery Form, which is shown to be immune to timing attacks but is much more complex. As for the representation for the points on the curve, simplicity and commonality is of consideration, therefore the chosen representation is affine coordinates. Also, by affine coordinates, the amount of temporary storage and extra registers needed are minimised. For the design of dual field multiplier, the basis chosen for GF(2^n) field must correspond to that of GF(p) for straightforward implementation. Therefore, polynomial basis is the most suitable for the purpose of this design.

The next section will introduce some of the most basic elliptic curve operations in GF(p) and GF(2^n).

3.3 Elliptic Curve Operations

In this section, the general elliptic curve operations will be introduced; they include point addition and point multiplication. Examples will be first given in elliptic curve over real number plane, over prime fields and finally in primary binary field. The examples given in this chapter are taken from [80].

For real number plane, the simpler form of the equation is the same as equation 3.3 and shown again as follows:

$$y^2 = x^3 + ax + b$$

where a and b are real numbers.

3.3.1 Point Addition over real number plane

The point P on the curve is represented as $P = (x, y)$ and its reflection in the x-axis forms its negative $-P = (-x, y)$. In order to add point P with Q, a straight line is drawn through the two points. The point where this straight line intersects the curve gives the reflection in the x-axis of the result, as demonstrated in Figure 3.1. However, the addition of $P + (-P)$, the line does not intersect with a third point and it is a vertical line, which provides the point of infinity O. $P + (-P) = O$ because P and $-P$ are reflection of each other in the x-axis as mentioned previously. The properties of the point at infinity were described in section 3.2.

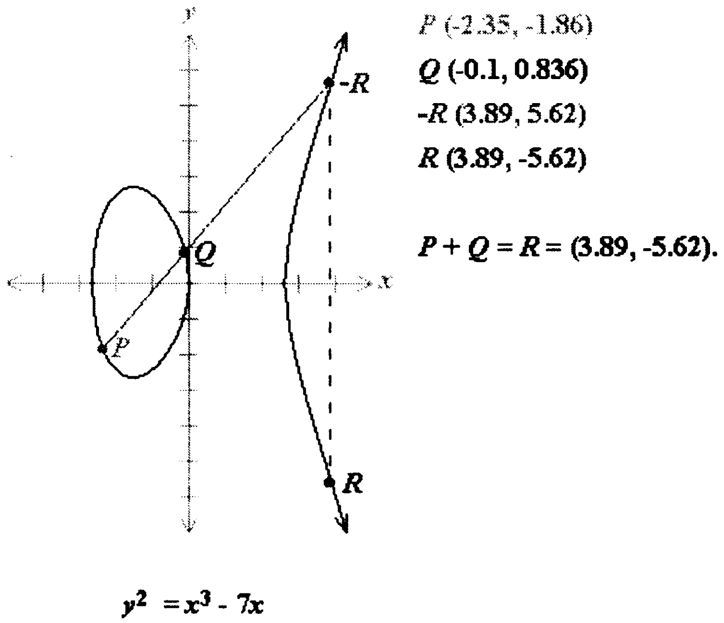


Figure 3.1 P+Q=R

Given the point P = (x_p, y_p) and Q = (x_q, y_q) and they are not negative of each other, then $P + Q = R$ could be obtained by first of all finding the slope of the intersecting line as follows:

$$s = \frac{y_p - y_q}{x_p - x_q} \tag{3.35}$$

The coordination of R = (x_r, y_r):

$$x_r = s^2 - x_p - x_q \tag{3.36}$$

$$y_r = -y_p + s(x_p - x_r) \tag{3.37}$$

3.3.2 Point Doubling over real number plane

Point doubling on an elliptic curve group is defined as follows:

$$2P = P + P = R \qquad (3.38)$$

The approach to perform point doubling, which is essentially adding the point to itself, is to draw a tangent on the point *P*, instead of drawing a line through two points *P* and *Q*. If the *y*-coordinate of *P* is not 0, i.e. it is not on the x-axis, then the tangent will intersect the curve at one other point and that point is the reflection in the x-axis of the result. This is demonstrated in Figure 3.2. In the case where the *y*-coordinate of *P* is 0, the tangent is a vertical line and it does not intersect the elliptic curve at other point, i.e. 2*P* = *O*.

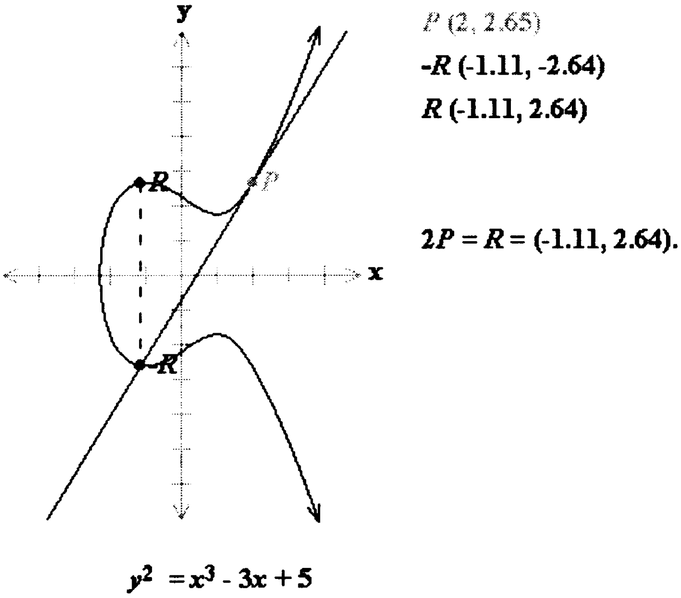


Figure 3.2 2*P*=*R*

Given the point $P = (x_p, y_p)$, $y_p \neq 0$ and *a* which is a parameter chosen when the elliptic curve is first set up. To calculate 2*P* = *R*, the tangent needs to be found first.

The tangent of the point *P* is given by:

$$s = \frac{3x_p^2 + a}{2y_p} \qquad (3.39)$$

The coordination of R = (x_r, y_r):

$$x_r = s^2 - 2x_p \tag{3.40}$$

$$y_r = -y_p + s(x_p - x_r) \tag{3.41}$$

3.3.3 Point Addition over GF(p)

Elliptic curve over GF(p) is defined as follows:

Let $F(p)$ be a finite field with $p > 3$, the elliptic curve E over the field GF(p) can be described by the short Weierstrass equation as shown in equation 3.3 :

$$y^2 = x^3 + ax + b$$

where a, $b \in$ GF(p). If $x^3 + ax + b$ contains no repeating factors or, equivalently, if the inequlity ($4a^3 + 27b^2$) mod p $\neq 0$, then the elliptic curve can be used to form a group. The elliptic curve group over GF(p) includes all points (x,y) which satisfy the elliptic curve equation (Equation 3.3) modulo p, where x and $y \in$ GF(p), together with the point at infinity O. There are finite number of points on such an elliptic curve.

The following shows an example of elliptic curve over GF(p) field F_{23}. With $a = 1$ and $b = 0$, Equation 3.3 becomes $y^2 = x^3 + x$ and the following 23 points satisfy the curve and are represented in Figure 3.3. Note that there is symmetry about $y = 11.5$. Over the field of F_{23}, the negative components in the y-values are taken modulo 23, resulting in a positive number with {0, ..., 22}.

(0,0)	(1,5)	(1, 18)	(9, 5)	(9, 18)	(11, 10)	(11, 13)	(13,5)
(13, 18)	(15, 3)	(15, 20)	(16, 8)	(16, 15)	(17, 10)	(17, 13)	(18, 10)
(18, 13)	(19, 1)	(19, 22)	(20, 4)	(20, 19)	(21, 6)	(21, 7)	

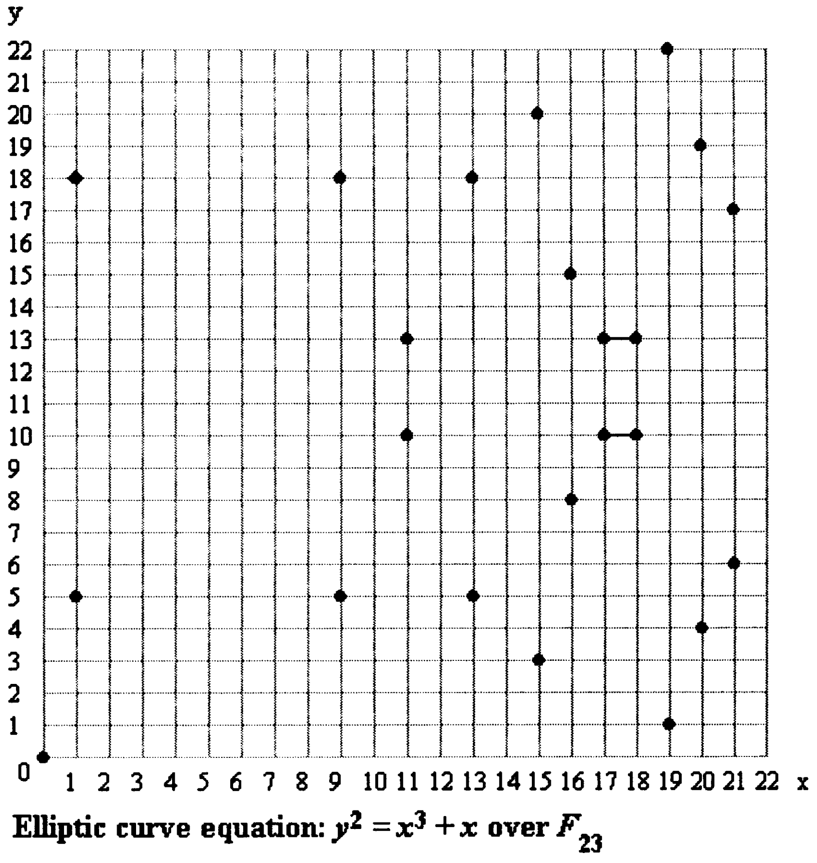


Figure 3.3 $y^2 = x^3 + x$ over GF(p) field F_{23}

The algebraic rules for the arithmetic are adapted for calculations of elliptic curves over GF(p). Given the points $P = (x_p, y_p)$, $-P = (x_p, -y_p \bmod p)$ and $Q = (x_q, y_q)$ and point $P \neq Q$, then $P + Q = R$ is calculated as follows:

$$s = \frac{y_p - y_q}{x_p - x_q} \bmod p \qquad (3.42)$$

The coordination of $R = (x_r, y_r)$:

$$x_r = s^2 - x_p - x_q \bmod p \tag{3.43}$$

$$y_r = -y_p + s(x_p - x_r) \bmod p \tag{3.44}$$

3.3.4 Point Doubling over GF(p)

Given the point $P = (x_p, y_p)$ and is $y_p \neq 0$, to calculate $2P = R$:

$$s = \frac{3x_p^2 + a}{2y_p} \bmod p \tag{3.45}$$

$$x_r = s^2 - 2x_p \bmod p \tag{3.46}$$

$$y_r = -y_p + s(x_p - x_r) \bmod p \tag{3.47}$$

3.3.5 Point Addition over GF(2^n)

Elliptic curve over GF(2^n) is can be defined as follow:

$$y^2 + xy = x^3 + ax^2 + b \tag{3.48}$$

where $a, b \in$ GF(2^n) and $b \neq 0$.

The elliptic curve group over GF(2^n) includes all points (x,y) which satisfy the elliptic curve equation (Equation 3.48), where x and $y \in$ GF(2^n) , together with the point at infinity O. There are finitely many points on such an elliptic curve.

An example of elliptic curve over GF(2^n) field $F_2{}^4$ is shown as follows and it is defined by using polynomial representation with the irreducible polynomial $f(x) = x^4 + x + 1$.

Given the element $g = (0010)$ is the generators for the field, the powers of g are:

$g^0 = (0001)$ $g^1 = (0010)$ $g^2 = (0100)$ $g^3 = (1000)$ $g^4 = (0011)$ $g^5 = (0110)$

$g^6 = (1100)$ $g^7 = (1011)$ $g^8 = (0101)$ $g^9 = (1010)$ $g^{10} = (0111)$ $g^{11} = (1110)$

$g^{12} = (1111)$ $g^{13} = (1101)$ $g^{14} = (1001)$ $g^{15} = (0001)$

Consider the elliptic curve:

$$y^2 + xy = x^3 + g^4x^2 + 1 \qquad (3.49)$$

Note that a in equation 3.48 has been substituted with g^4 and b with $g^0 = 1$. The fifteen points that satisfy the equations are shown as follows, these points can be depicted in a graph as shown in Figure 3.4.

$(1, g^{13})$ (g^3, g^{13}) (g^5, g^{11}) (g^6, g^{14}) (g^9, g^{13}) (g^{10}, g^8) (g^{12}, g^{12})

$(1, g^6)$ (g^3, g^8) (g^5, g^3) (g^6, g^8) (g^9, g^{10}) (g^{10}, g) $(g^{12}, 0)$ $(0, 1)$

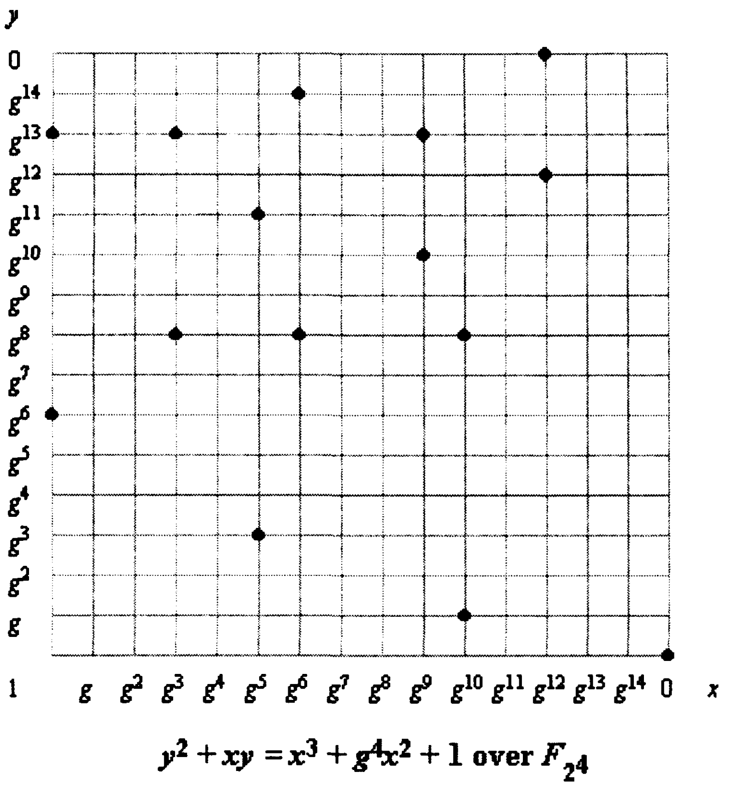


Figure 3.4 $x^3 + g^4x^2 + 1$ over F_2^4

The algebraic rules for the arithmetic are adapted for calculations of elliptic curves over F_2^m. Given the points $P = (x_p, y_p)$, $-P = (x_p, x_p + y_p)$ and $Q = (x_q, y_q)$ and point $P \neq -Q$, then $P + Q = R$ is calculated as follows:

$$s = \frac{y_p - y_q}{x_p - x_q} \tag{3.50}$$

$$x_r = s^2 + s + x_p + x_q + a \tag{3.51}$$

$$y_r = s(x_p + x_r) + x_r + y_p \tag{3.52}$$

The properties of the point of infinity that is shown in the real number plane case also applies to GF(2^n), such that, $P + (-P) = O$ and $P + O = P$.

3.3.6 Point Doubling over GF (2^n)

Given the point $P = (x_p, y_p)$ and is $y_p \neq 0$, to calculate $2P = R$:

$$s = x_p + \frac{y_p}{x_p} \tag{3.53}$$

$$x_r = s^2 + s + a \tag{3.54}$$

$$y_r = x_p + (s+1) * x_r \tag{3.55}$$

If $y_p = 0$, then $2P = O$.

3.4 ECC and Side Channel Attacks

In the first part of this section, some known attacks for solving the elliptic curve discrete logarithm problem (ECDLP) and the techniques to avoid these attacks will be explained. In the second part of this section, side channel attacks will be introduced. Instead of attacking the algorithm itself, side channel attacks are attacks against the cryptographic devices and their implementations.

3.4.1 Known ECDLP attacks

Some known weaknesses in elliptic curve cryptography algorithms will be explained in this section. The purpose of these attacks is to solve the elliptic curve discrete logarithm problem (ECDLP), which was described in section 2.6.1. Some of the points have been mentioned briefly previously in this chapter, more explanations will be given in this chapter. [81] and [82] provides a good overview on this topic.

1. Naive exhaustive search

   This method requires the attacker to compute successive multiples of P: P, $2P$, $3P$, $4P$… until the public key is obtained. This attack is impractical for high order cryptosystem.

2. Pohlig-Hellman algorithm [83]

   This attack exploits the factorization of the order of the point P, n. This algorithm reduces the complexity of recovering the discrete logarithm k of Q to the base P to the problem of recovering k modulo each of the prime factors of n, where k can then be recovered by using the Chinese Remainder Theorem. In order to construct the most difficult case of the ECDLP, the order of the elliptic curve chosen must be divisible by a large prime n, e.g., $n > 2^{160}$ bits. Preferably, this order should be a prime number or almost a prime, which means that a large prime number times a small integer.

3. Baby-step giant-step algorithm

   This attack is a time-memory trade-off of exhaustive search. Instead of the worst case of up to n steps required by traditional exhaustive search, only $\sqrt{}n$ steps in the worst case but requires memory for $\sqrt{}m$ points, where m denotes the memory size.

4. Pollard's Rho algorithm [84]

   This algorithm is generally regarded as the best general-purpose algorithm known for solving ECDLP [82]. This is essentially a randomized version of baby-step giant stop algorithm. The running time of this algorithm is very similar to that of baby-step giant-step algorithm, however, it requires less memory space. Teske [85] provided an improved version, which has an expected running time of $\sqrt{\pi n/2}$ and negligible storage requirements. This algorithm is most effective for factoring integers with small factors, therefore, this can be avoided by using a high order number.

5. Parallelized Pollard's Pho algorithm [86]

Van Oorschot and Wiener described the method to parallelise the Pollard's Rho algorithm. When the algorithm is run in parallel using r processors, it results in an r-fold speed up of roughly $\sqrt{\frac{\pi n}{2r}}$ steps.

6. Multiple logarithms [87]

Silverman and Stapleton suggested that successive logarithms become easier to solve once the first instance of the ECDLP has been worked out. The method to avoid this occurring is to ensure that the elliptic curve parameters are chosen so that the first instance is infeasible to solve.

7. Supersingular Elliptic curves [66] [88] [89]

Supersingular curve is an elliptic curve $\boldsymbol{E}$ over $\boldsymbol{Fq}$ where the trace t of $\boldsymbol{E}$ is divisible by the characteristic p of $\boldsymbol{Fq}$. It is known that supersingular curves are of some extension field $\boldsymbol{F}_q{}^k$ where $k \leq 6$ and a subexponential-time algorithm exists for the ECDLP in singular curves.

In general, under mild assumptions, the ECDLP in an elliptic curve $\boldsymbol{E}$ defined over a finite field $\boldsymbol{F}_q$ can be reduced to the ordinary DLP in the multiplicative group of some extension field $\boldsymbol{F}_q{}^k$ for some $k \geq 1$, where the number field sieve algorithm applies. This is known as the Weil and Tate pairing attacks. In order to ensure the reduction algorithm does not apply to a particular curve, the order n of the point P should not divide $q^k - 1$ for all small k for which the DLP in $\boldsymbol{F}_q{}^k$ is tractable.

8. Weil Descent [90] [91] [92]

Weil descent is efficient for reducing the ECDLP in an elliptic curve $\boldsymbol{E}$ over a characteristic two finite field $\boldsymbol{F}_2{}^m$ to the discrete logarithm problem in the jacobian $J_c(\boldsymbol{F}_2{}^n)$ of an algebraic curve $\boldsymbol{C}$ defined over a subfield $\boldsymbol{F}_2{}^n$ of $\boldsymbol{F}_2{}^m$ [91] .

Let $k = \boldsymbol{F}_q$ denote some finite field of characteristic two, and let $n \geq 2$ denote an integer, where n is quite small and q is large such that $q^n > 2^{160}$ in practice. Let K denote the field extension $\boldsymbol{F}_q{}^n$, with k-basis $\{\psi_0, \psi_1, \ldots, \psi_{n-1}\}$. Given an elliptic curve E over K:

$$Y^2 + XY = X^3 + \beta \qquad (3.56)$$

where $\beta \in$ K. Assume that $\boldsymbol{E}(\boldsymbol{F}_q{}^n)$ contains a subgroup of prime order p with $p \approx q^n$.

$$\beta = b_0\psi_0 + b_1\psi_1 + \ldots + b_{n-1}\psi_{n-1} \qquad (3.57)$$

$$X = x_0\psi_0 + x_1\psi_1 + \ldots + x_{n-1}\psi_{n-1} \qquad (3.58)$$

$$Y = y_0\psi_0 + y_1\psi_1 + \ldots + y_{n-1}\psi_{n-1} \qquad (3.59)$$

By substituting Equation 3.57, 3.58 and 3.59 into Equation 3.56 and equating coefficients of ψ_i, an abelian variety A define over k of dimension n is obtained. The abelian variety A is called the Weil restriction and the process shown above, where the abelian variety A could be achieved, is called Weil decent.

Gaudry Hess and Smart [92] gave an explicit algorithm for the case where the algebraic curve C is a hyperelliptic curve of genus g defined over $\boldsymbol{F}_2{}^n$. The variation of attack is known as GHS attack.

In order to prevent these attacks, the use of elliptic curves over finite fields F_2^m where m is composite should be avoided.

9. Prime field anomalous curves

[93] [94] [95] showed ECDLP can be solved efficiently for prime field anomalous curves where the number of point of an elliptic curve $\boldsymbol{E}$ over $\boldsymbol{F_p}$, $\#\boldsymbol{E}(\boldsymbol{F_p})$ is equal to p. Therefore, the number of points on an elliptic curve must not equal to the cardinality of the underlying field

10. Hyperelliptic curves

Hyperelliptic curves are a family of algebraic curves of arbitrary genus that includes elliptic curves, therefore an elliptic curve is effectively a hyperelliptic curve of genus 1. The definition of hyper elliptic curve is as follows:

Let $\boldsymbol{F_q}$ be a finite field. A hyperelliptic curve C of genus g over $\boldsymbol{F_q}$ ($g \geq 1$) is a non-singular curve given by an equation of the form:

$$y^2 + h(x)y = f(x) \qquad (3.60)$$

where $h(x) \in \boldsymbol{Fq}[x]$ is a polynomial of degree $\leq g$ and $f(x) \in \boldsymbol{Fq}[x]$ is a monic polynomial of degree $2g$+1.

Adleman, DeMarrais and Huang [96] presented a subexponential-time algorithm for DLP in the jacobian of a large genus hyperelliptic curve over a finite field of prime characteristic. Enge [97] provided a subexponential algorithm for solving the discrete logarithm problem in Jacobians of high-genus hyperelliptic curves over any finite fields. Therefore high-genus hyperelliptic curves should be avoided.

11. Non-applicability of index-calculus methods

No general subexponential-time algorithm has been discovered yet. [53] and [98] provided arguments for why the index-calculus algorithms may be applicable to the ECDLP.

In summary, the general methods to avoid these attacks are to avoid using certain known curves and also the size of the modulus should abide to the recommended minimal size (see Table 2.3).

3.4.2 Side channel attacks

Side channel attacks were first proposed by Kocher [99] in 1996. Unlike the attacks that were mentioned previously, which are based on information on the plaintext or the ciphertext, side channel attacks are based on measurable side channel information that can be retrieved, such as computation time and power consumption. By making use of these side channel information, the attacker can deduce the inner-working algorithm of the system and so some secret information needed, such as the secret key. There are two different classes of side channel attacks depending on side channel information retrieved and they are:

- Simple side channel attacks, where only a single measurement is needed

- Differential side channel attacks, where several measurements handled together with statistical tools to correlate the secret information with the collected data

In this section, some of the common side channel attacks will be described, also the techniques required to defend against these attacks will also be introduced.

3.4.3 Simple Side Channel Analysis

Two of the most common simple side channel analyses are:

1. Timing attack

   Timing attack was first introduced by Kocher [99] . Timing attacks are based on measuring the time taken for the system to perform certain cryptographic operation.

2. Simple Power Analysis

   Simple power analysis (SPA) was discussed in [100]. SPA is based on analysing the power consumption of the device during operation. Since the integrated circuits are made up of many transistors, the charging and discharging activity of each transistor while switching can be detected. The method to measure the power consumption of a device is to simply connect a small resistor, e.g. 50 ohm, in series with the power or the ground input, the power consumption can be calculated simply by computing I (current) = V (voltage)/ R (resistance).

In the case of elliptic curve cryptography, one of the most common techniques for computation of a scalar multiplication is the double-and-add method, which is described in Figure 3.5. It is very common that input P is public and the scalar k is secret, therefore k is of interest to the attacker.

```
Algorithm: double-and-add

Input: $P$, $k=(1, k_{l-2}, \ldots, k_0)_2$
Output: $kP$

$Q := O$
For $i$ from $n$-1 down to 0 do
$Q := 2Q$
if $k_i = 1$ then
        $Q := Q + P$
return $Q$
```

Figure 3.5 Double-and-add method

Based on this common algorithm, the attacker can attempt to determine important information by performing simple timing analysis or simple power analysis on the system at the if-branch (see line 4 of Figure 3.5). Since that point doubling happens in every iteration, but point addition is only executed when jth bit of k is equal to 1, the attacker may be able to distinguish the two operations based on the information collected. [101] provided suggestions on parallel scalar multiplication on general elliptic curves over $\boldsymbol{F_p}$ hedged against non-differential power attacks, by using the Montgomery Ladder algorithm (See Section 3.4.5 for more details).

3.4.4 Differential Power Analysis (DPA) attack

One of the most common differential power attacks is differential power analysis [100]. Similar to SPA, DPA requires the knowledge of power consumption of system to obtain desired knowledge of the cryptosystem, however, unlike SPA where only a single measurement is involved, DPA consists not only of study of various visual data sample, but also statistical analysis and error correction statistical methods. DPA is such a powerful attack that it can automatically locate correlated regions in a device's power consumption, the attack can be automated and little or no information on the implementation of the system is required.

The attacker observes m encryption operations and captures some powers traces where each trace should contain *k* samples each and record the corresponding ciphertext. The attacker will then be able to use the information on power consumption from the power traces obtained and statistical methods to determine the secret information. For example, DPA attacks on DES round one can be carried out as follows [162]: recall the S-Box that exists within the DES function as shown in Figure 2.4, each of these box is analysed one at a time by DPA. Differential power consumption curves (PCCs) of the subject are then collected. The PCCs are then grouped together in to calculate a differential curve. The attack will have to perform the partial traces calculation for each of the 2^6 6-bit partial subkey combinations. The correct subkey can be known by looking for the curve that is formed by the correct subkey. Coron [102] presented a general overview of resistance against DPA for ECC, summarised as follows:

3.4.5 Countermeasure Against Side Channel Attacks

1 General Countermeasures

In terms of hardware implementation, smart card is particularly prone to side channel analysis attacks, so tamper-resistant smart card should be used. Kömmerling and Kuhn [103] provided a good overview on smart card technology against side-channel attacks.

Secondly, system parameters and inputs should be validated first since there are some attacks that attack the system by feeding predefined special [104] or erroneous input to the algorithm [105], or to provoke faults in the process [106].

2 Countermeasures against Simple Analysis Attacks

The main approach to defend simple analysis attacks is to achieve uniform execution pattern. This can be realized by the following methods:

- Using an algorithm where the order of operations is fixed, so that the operations become indistinguishable because of their regular occurrences. [102] [104] [107]

- Reconstruct the common double-and-add algorithm so that the same field arithmetic operations are used to disguise the differences of the two operations. [108] [109] [110]

- Used random values to split or mask the secret data. [102]

As mentioned at the beginning of this chapter, Montgomery Form is secure against timing attacks [60]. Since the time required to perform the conventional scalar multiplication algorithm based on the Weierstrass-form depends on the bit patterns of the secret value (and on the ratio between the number of zeros and ones), systems based on Weierstrass-form are insecure against timing attacks.

However, this is not the case for systems based on Montgomery Form. It has exactly seven multiplications and four square-multiplications on F_p per bit. Reference [61] also suggested the use of the randomised projective coordinate representation increases the difficulty to measuring the timing of the algorithm. This approach has a limitation to specially chosen curves, in this case, curves with Montgomery form of group order divisible by 4.

Another method to defend simple analysis attacks is to use Montgomery ladder [61] [101]. The Montgomery Ladder was initially designed to accelerate the scalar multiplication on a restricted class of curves over Fp. Reference [111] provides a good overview on Montgomery Ladder.

Figure 3.6 shows the Montgomery Ladder algorithm. Let k be a positive integer and $(k_{n-1}, \ldots, k_{n-1-i})$ its binary representation. Initially we have the pair $(P, 2P)$ and at the beginning of each iteration, the pair $(P_1, P_2) = (mP, (m+1)P)$, where $m = (k_{n-1}, \ldots, k_{n-1-i})$. The final result is $(kP, (k+1)P)$.

```
Algorithm: The Montgomery Ladder

Input: $P$, int $k \geq 1$
Output: $kP$

$P_1 := P$ and $P_2 := 2P$
For $i$ from $n$-1 down to 0 do
if $k_i = 1$ then
        $P_1 := P_1 + P_2$ and $P_2 := 2P_2$
else
        $P_2 := P_1 + P_2$ and $P_1 := 2P_1$
return $P_1$
```

Figure 3.6 The Montgomery Ladder

Coron [102] presented a revised double-and-add algorithm so that the operation in the algorithm, such as the branching operation, can not be identified; this is shown in Figure 3.7. This technique works based on insertion of dummy instructions.

```
Algorithm: Double-and-add resistant against SPA

Input: $P$, int $k \geq 1$
Outout: $kP$

$Q_0 := P$
For $i$ from $n$-1 down to 0 do
$Q_0 := 2Q_0$
$Q_1 := Q_0 + P$
$Q_0 := Q_i$
Return $Q_0$
```

Figure 3.7 Double-and-add resistant against SPA

3 Countermeasures against Differential Analysis Attacks

The main approach to defend simple analysis attacks is to diminish the correlation between any assessment results. This can be achieved by randomizing the values of the base point ***P*** and any intermediate points involved in a calculation. This can be realized by the following methods:

- Reference [112] and [113] proposed the use of randomized algorithms, which could obscure the correlation of the intermediate results.
- Reference [114] introduced the idea of disguising the elliptic curve or the field by replacing it with random, isomorphic versions
- Reference [102] also contributed toward defence DPA. Coron introduced three countermeasures:
    - Randomisation of the private component
    - Blinding the point *P* by adding a secret random point *R*
    - Randomisation of the representation projective coordinates by introducing a random scaling factor.

Walter [1] suggested that the longer the key length the greater the side channel leakage, even with the increase in mathematical strength in the cryptosystem. Therefore with longer key, it could actually mean lower security, because of the greater number of arithmetic operation needed leading to greater leakage. The fact that ECC can provide similar level of security compared with RSA with a much shorter key length (see Table 2.3), ECC is better option in terms of resistance to side channel attacks.

3.5 Literature Review

There have been many different hardware designs for different arithmetic processes for the fields GF(p) and GF(2^n). For example, Guajardo *et. al.* described a hardware implementation of a modulo multiplier for GF(p) based on Residue Number System

(RNS) [115]. Gutub *et. al.* [116] presented a scalable VLSI architecture for GF(p) Montgomery modular inverse computation. Scalability implied the flexibility of having an arbitrary operand size, such that the same design of hardware can be reused to expand the size of the operand, but at the same time the logic depth of the structure is unchanged. This is important for cryptographic hardware, since the mathematical difficulty of the cryptosystem lies in the length of the key and the requirements of key length varies. Also, Wu [117] presented a bit-parallel Montgomery multiplier design for GF(2^n) for modulo multiplication and squaring that use m^2 gates.

The use of Montgomery multiplication [118] is very popular because of the simple modulo reduction operation used. Montgomery multiplication makes use of the least significant digit of an accumulating product to determine the multiple of M to subtract. Another example of a scalable modular multiplier based on Montgomery multiplication is presented by Tenca and Koc [119]. Walter [120] provided an overview on techniques for the hardware implementation of Modular Multiplication, more details on the implementation of Montgomery multiplication will be presented in Chapter 4.

Savaş *et. al.* proposed the first hardware implementation of a unified field multiplier which can operate on both GF(p) and GF(2^n) [121], [122]. This design makes use of a LSB-first word-serial Montgomery multiplication and the operands are required to be transformed into the Montgomery domain.

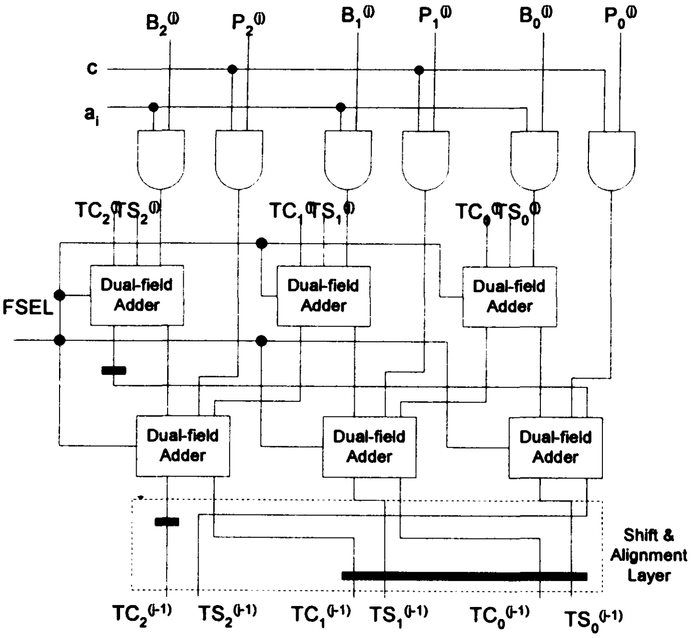
Word-serial architecture is similar to "pen and paper" method such that to compute $A \times B$ where $A = \{a_i, a_{i-1}, \ldots a_1, a_0\}$ and $B = \{b_i, b_{i-1}, \ldots b_1, b_0\}$:

```
*Algorithm: pen and paper multiplication*

Input: $A, B$
Outout: $C = A*B$

$C := 0$
For $n$ for from $i$ down to 0 do
$C_i := A * b_n$
$C_i := C_i * 2^{i-1} + C_{i-1}$
Return $C$
```

Figure 3.8 Pen and paper multiplication

Bit-parallel means that instead of having A multiplied by each bit of B (b_i) to give individual values of Ab_i one cycle at a time, all the Ab_i values are generated at once and then summed up. The disadvantage of bit-serial is that the process is very slow however it is very simple and easy to implement; the disadvantage of bit-parallel is it is area consuming.

The processing unit of Savaş' design is shown in Figure 3.9.



B_t^j = multiplicand B t^{th} bit j^{th} iteration
P_t^j = muodulus P t^{th} bit j^{th} iteration
a_i = multiplier a i^{th} bit
c = carry
$FSEL$ = field selection
($TC^{(j)}$, $TS^{(j)}$) = partial sum T expressed in redundant carry save form
■ = pipeline latches

Figure 3.9 Savaş' et. al. Processing Unit: wordlength = 3

The selection of the field is done by setting the *FSEL* input. When *FSEL* = 1, the system performs operation over GF(p) and when *FSEL* = 0, the system runs in the GF(2^n) mode. Figure 3.10 shows the dual-field adder circuit synthesised for this implementation. Note that in this dual field circuit, the critical path from any data input (a, b or c) to either output traverses four logic levels, assuming XOR gates have a logic depth of 2.

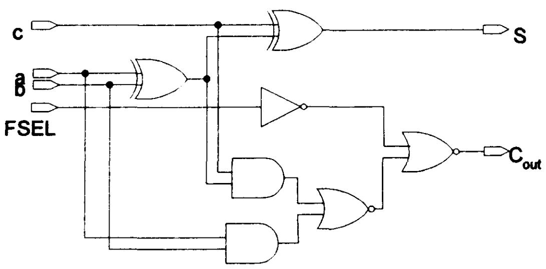


Figure 3.10 Savaş' dual-field adder synthesised by Mentor

Johann Groβshädl [123] proposed a bit-serial unified multiplier architecture for finite field GF(p) and GF(2^n) in 2001 based on an MSB-first iterative algorithm for modulo multiplication. Figure 3.11 shows the arithmetic unit that is used for the implementation of the modular multiplier. The first (n+1)-bit carry-save adder performs the addition of the partial products. The output *Sum* R_s and *Carry* R_c are used to estimate the multiple of the modulus to be subtracted in the next step with another (n+1)-bit carry-save adder.

Figure 3.12 is a block diagram of Groβshädl's bit-serial multiplier architecture. In order to perform carry-free addition for GF(2^n), all the carry bits of the adder (R_c) are set to 0, which in turn set further control signals. Modulo reduction occurs within the multiplication process by concurrent subtraction of a multiple of the modulus.

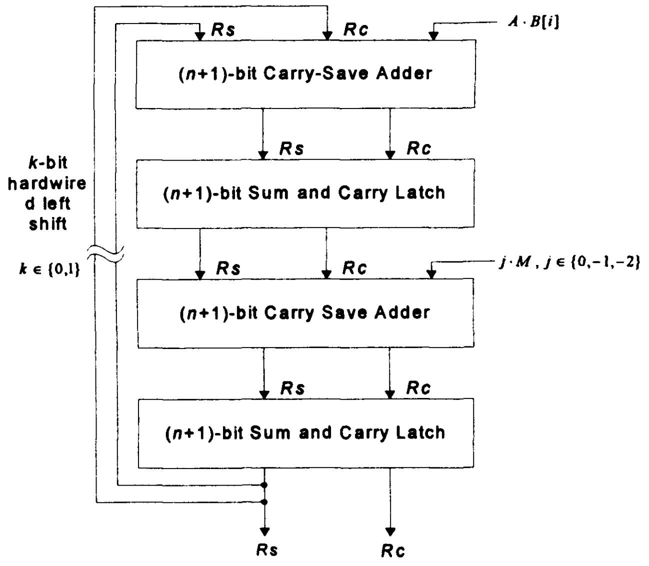

Rs
Rc
$A \cdot B[i]$
(n+1)-bit Carry-Save Adder
Rs
Rc
k-bit hardwired left shift
$k \in \{0,1\}$
(n+1)-bit Sum and Carry Latch
Rs
Rc
$j \cdot M, j \in \{0,-1,-2\}$
(n+1)-bit Carry Save Adder
Rs
Rc
(n+1)-bit Sum and Carry Latch
Rs
Rc

Figure 3.11 Arithmetic unit of Groβshädl's n-bit unified multiplier

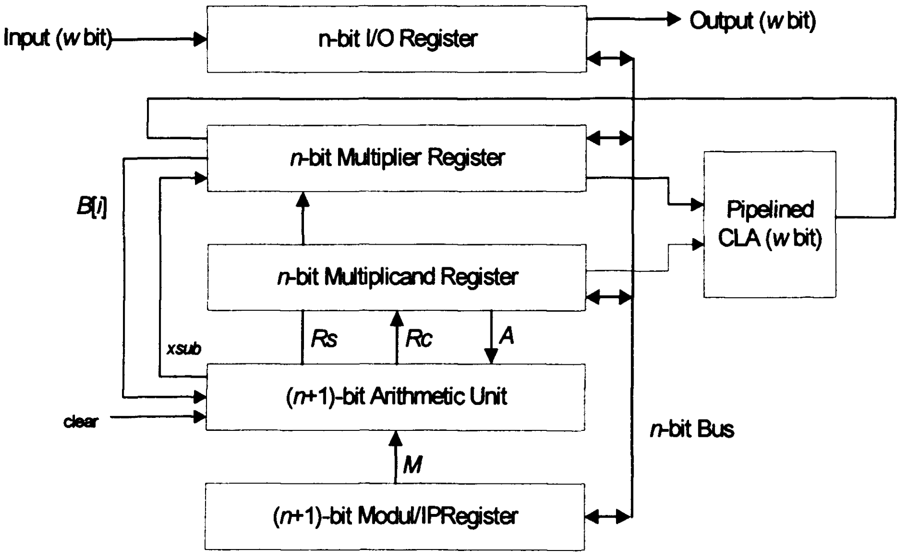

Input (w bit)
n-bit I/O Register
Output (w bit)
n-bit Multiplier Register
B[i]
Pipelined CLA (w bit)
n-bit Multiplicand Register
xsub
Rs
Rc
A
(n+1)-bit Arithmetic Unit
clear
n-bit Bus
M
(n+1)-bit Modul/IPRegister

Figure 3.12 Block diagram of Groβshädl's bit-serial multiplier architecture

Wolkerstorfer has also proposed a dual-field arithmetic unit for GF(p) and GF(2^n) [124]. Figure 3.13 shows the carry-save adder used in his design. Dual-field architecture designs also exist for field inversion ([125] and [126]).

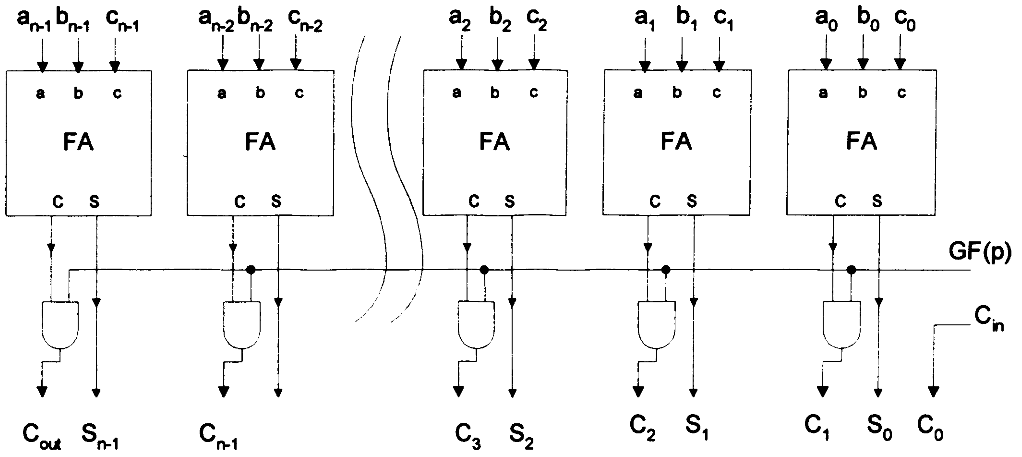


Figure 3.13 Carry Save Adder in Wolkerstorfer's design

All these proposals require broadcasting a control signal to all the full adder cells to force the output carries low from all the full adders in the multiplier. This is costly and slow, especially when switching between fields, as can occur often in a server operating on many different data streams. This thesis describes a new multiplier which operates in both GF(*p*) and GF(2^n), and makes use of a novel dual field adder based on a (4:2) carry-save adder cell, modified so that it is capable of adding specially-encoded operand digits [127], which will be described in chapter 4.

4 Unified Field Redundant Adder

Chapter 3 demonstrated that all the previous dual-field designs required the field information signal to be broadcasted throughout the adder structure, which is costly and slow due to high signal fan-out. In this chapter, the implementation of the proposed unified field redundant adder, which is required for the overall implementation of the unified field multiplier, will be described.

The design of this unified field adder can perform addition in both prime and binary field without compromising the performance and area requirement compared with conventional adders and previous designs. The main difference between the proposed design and the previous designs is that, in the proposed design, the field information is embedded within the encoding itself, such that field information signal is not broadcasted throughout the module. The proposed design is impartial such that addition in either field can be carried out equally easily. Furthermore, this design is scalable, where the wordlength can be scaled up or down by reusing, replicating or truncating the adder modules, without affecting the logic depth of the hardware structure.

4.1 Truly scalable unified field redundant adder

Many of the existing unified multiplier designs require an external field selector signal to choose between the two fields - $GF(p)$ and $GF(2^n)$. This external signal is fed into the adder gates to force the carries low in $GF(2^n)$, however, this also means that this one signal has very high fan-out especially for large multiplication which is often the case in cryptography. This can be demonstrated by the processing unit of the multiplier proposed by Savas [121], which is shown in Figure 3.9, where one can see that for a processing unit of word length = 3 bits, the field select signal has to drive 6 gates. Therefore for an n-bit processor, the field select signal will have to drive 2-n bits. This method is inefficient because the field select signal will need to have a very high drive strength which could also affect the scalability of the overall multiplier since buffers will need to be added.

The proposed method to avoid this problem is to incorporate the field information into the representation of the number itself.

4.1.1 Redundant Number Representation and redundant adder

The Redundant Binary Adder, illustrated in Figure 4.1, is a binary adder capable of adding two numbers with the digit set $d_i \in \{0, 1, 2\}$ (or equivalently $d_i \in \{-1, 0, 1\}$) such that carry bits do not propagate over the whole length of the sum [128]. Note that carry signals transform from {0,2} to {0,1} as they are carried forward to the next bit. Each block in the first two rows of Figure 4.1 can be implemented as a full adder as shown in Figure 4.2. The last row of blocks simply merges pairs of inputs to provide the output digits.

In the Redundant Binary Adder, digits are implemented using two binary signals. If neither signal is 'High' the value '0' is represented; if both signals are 'High' the value '2' is represented; otherwise, if either one of the signals is 'High' the value '1' is represented as shown in Table 4.1. A variety of other coding schemes are possible, but similar to the example just given, only 3 signal combinations are needed to represent 3 values, i.e. there will always be a redundant set of combination. The advantage of redundant adder is that it can avoid long carry chain and has a constant delay, which is independent of the adder width. Additionally, the area cost is directly proportional to the word length *n*. (4:2) redundant adders have been used before in binary multiplier designs as an alternative to carry-save adders because they have more regular multiplier tree layouts, requiring less interconnect than other reduction tree topologies [129], [130]. The new adder will make use of these same layout advantages to be applied to Galois Field multiplication under either GF(p) or GF(2^n). The schematic diagram of the conventional (4:2) redundant adder is shown in Figure 4.1. In conventional redundant adder design, the states '0', '1' and '2' are represented by two wires as shown in Figure 4.1, therefore, in conventional (4:2) redundant adder design, the state '1' is represented by both '01' and '10'.

Table 4.1 Conventional Redundant Representation

wires	Number representation
00	0
01	1
10	1
11	2

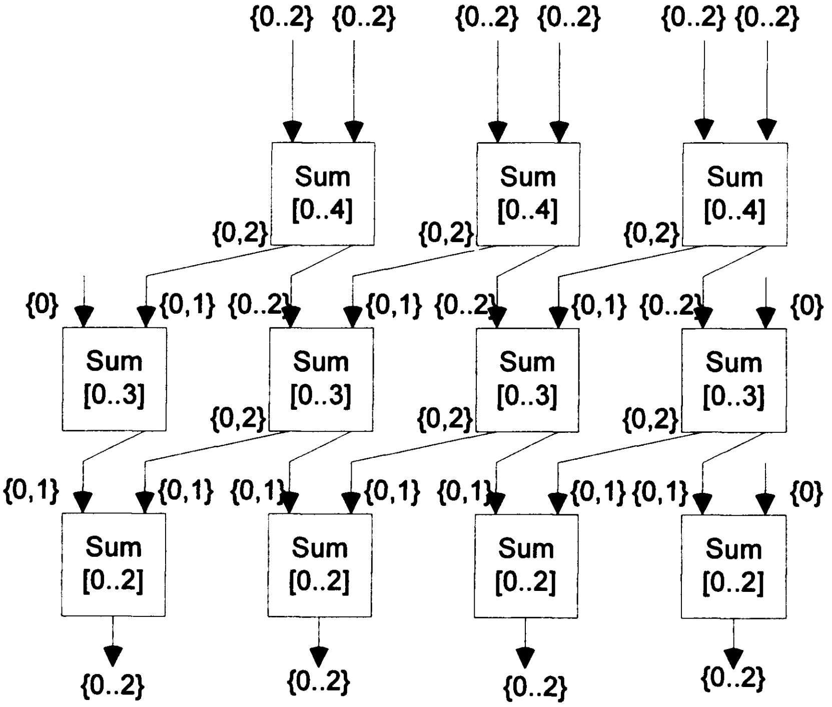


Figure 4.1 Conventional Redundant Adder $w = 6$

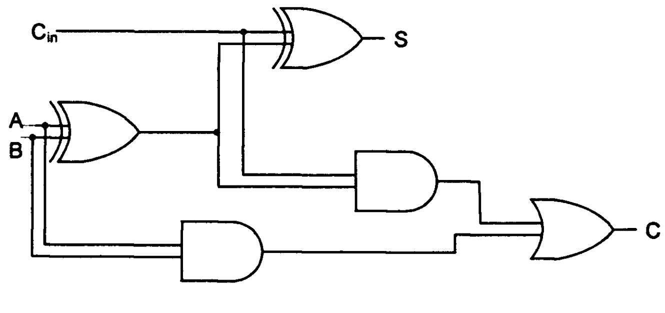


Figure 4.2 Binary Full Adder

4.2 Unified field Redundant Adder

A dual-field Galois Field adder can be constructed by introducing a fourth digit value, denoted 1*, that indicates the digit '1' over GF(2^n). Hence, addition over GF(p) is implementable using the digits {0,1,2}, while addition over GF(2^n) is implementable using the digit set{0,1*}. The characteristic of addition over GF(p) is depicted in Table 4.2. Addition over GF(2^n) can be characterised by the expressions: 1* + 1* = 0, and $0 + k = k + 0 = k$ and is shown in Table 4.3. The digit sets for addition in the two fields can be defined as follows:

- for GF(p), 3 values are needed: {0, 1, 2}
- for GF(2^n), only 2 values are needed: {0, 1*}

Therefore, 5 values are apparently required in total. However, only 4 values are actually needed because the zero elements in both fields are defined identically, such that 0 + 0 = 0 in both cases (see Table 4.2 and Table 4.3).

Table 4.2 Table of addition for GF(p)

	0	1	2
0	0	1	2
1	1	2	3
2	2	3	4

Table 4.3 Table of addition of for GF(2^n)

	0	1*
0	0	1*
1*	1*	0

By incorporating the 1* digit into the Redundant Binary Adder (Figure 4.1), a dual Galois Field adder can be formed with little adjustment, as shown in Figure 4.3. This enables us to take advantage of previously unexploited "don't care" states in the (4:2) adder cell. The four symbols {0,1,2,1*} require two wires for their full representation, similar to the redundant binary adder of Figure 4.1. The unified field adder will be structurally very similar to the Redundant Binary Adder (Figure 4.1), however, the cells are not now full adders, and so optimum logic circuits for the dual field adder need to be derived.

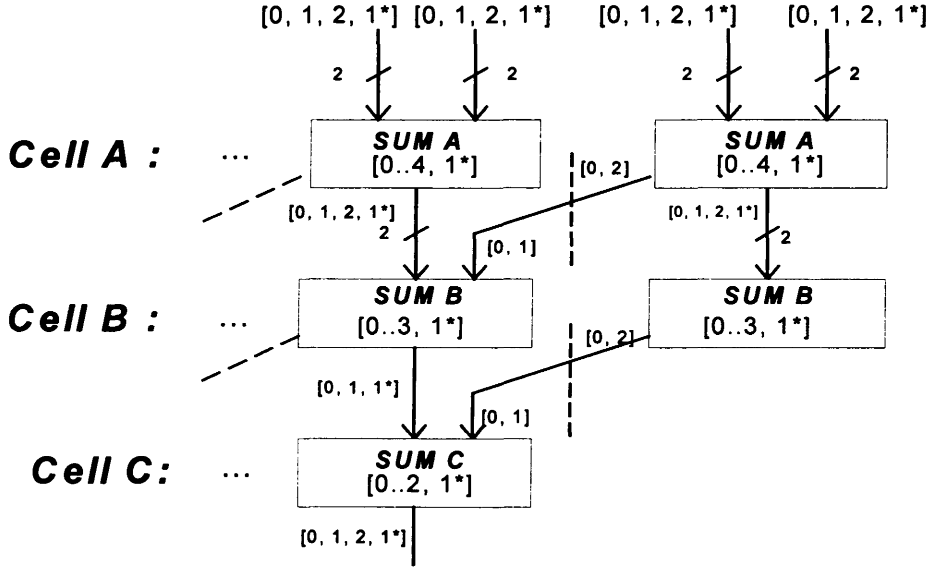


Figure 4.3 Redundant Dual Field adder

The new (4:2) adder comprises three separate stages, implemented using three different cells: the first stage (cell A) receives two 2-bit operands, $x(1:0)$ and $y(1:0)$ with the digit set, $d \in \{0,1,2,1^*\}$, and adds them to form a 2-bit sum digit, $S_A \in \{0,1,2,1^*\}$, and a carry bit, $C_A \in \{0,2\}$. The addition is summarised in Table 4.4, showing that there is much flexibility available for the cell's implementation. Don't care state is formed when a value in GF(p) is to be added to a digit in GF(2^n), since this is prohibited. Note that, the output digit '2', can be represented by either $S_A = 2$ and $C_A = 0$ or $S_A = 0$ and $C_A = 2$.

Table 4.4 Cell A addition

	0	1	2	1*
0	0	1	2	1*
1	1	2	3	X
2	2	3	4	X
1*	1*	X	X	0

The second stage (cell B) receives the 2-bit sum digit, S_A, and the shifted carry bit, C_A, from the previous bit of cell A, and adds them to form the 2-bit sum digit, $S_B \in$ {0,1,1*}, and a carry bit, $C_B \in$ {0,2}. The addition is summarised in Table 4.5, showing that there is less flexibility available in this cell's implementation than in cell A, as there is only one don't care state.

Table 4.5 Cell B addition

	0	1
0	0	1
1	1	2
2	2	3
1*	1*	X

Finally, the third stage (cell C) receives the 2-bit sum digit, S_B, and the shifted carry bit, C_B, from previous bit of cell B and adds them to form the 2-bit sum digit, $S_C \in$ {0,1,2,1*}. This digit set matches the digit set of the (4:2) adder's inputs, so that the addition is complete [128]. The third stage of the addition is summarised in Table 4.6, showing that there is more flexibility available in this cell's implementation, due to the increased number of don't care states.

Table 4.6 Cell C Addition

	0	1
0	0	1
1	1	2
X	X	X
1*	1*	X

4.2.1 Cell A digit coding

The digit coding for cell A was chosen as follows: $S_A(0)$ should be a 2-input XOR function, to match the delay of a basic (4:2) adder, the other two logic functions are required to be as simple as possible. Having experimented with all possible coding combinations, the most efficient coding scheme was found to be (0,1) = 1*, and (1,1) = 1. (0,0) was chosen to be 0, leaving (1,0) = 2. Filling out Table 4.4 with these digit representations gives the Karnaugh map shown in Table 4.7, where '-' reflects that the decision about how to represent the output '2' is yet to be made, and 'X' denotes "don't care".

Table 4.7 Karnaugh Map for Cell A addition

	0 → 00	**1* → 01**	**1 → 11**	**2 → 10**
0 → 00	0, (0,0)	0, (0,1)	0, (1,1)	-, (-,0)
1* → 01	0, (0,1)	0, (0,0)	X, (X,0)	X, (X,1)
1 → 11	0, (1,1)	X, (X,0)	-, (-,0)	1, (1,1)
2 → 10	-, (-,0)	X, (X,1)	1, (1,1)	1, (1,0)

The '-, (-,0)' entries must become either '1, (0,0)' or '0, (1,0)' to represent an output value of 2 (see Table 4.8). If they are set to '0, (1,0)', then $S_A(1) = x(1) \vee y(1)$, using the don't care states. Finally, by setting all the remaining don't care states for C_A low, $C_A = x(1) \wedge y(1) \wedge \{\neg x(0) \vee \neg y(0)\}$ is obtained, which is implemented as a 2-input NAND driving a 3-input AND, matching the CMOS VLSI delay of the XOR. Note that $\neg$ denotes inversion, $\vee$ denotes 'OR' function and $\wedge$ denotes 'AND' function. The final map for cell A is presented in Table 4.9.

Table 4.8 Redundant representation of sums

Total	SUM	CARRY
0	0	0
1	1	0
2	2	0
2	0	1
3	1	1
4	2	1

Table 4.9 Karnaugh Map for Cell A addition

	0 → 00	**1* → 01**	**1 → 11**	**2 → 10**
0 → 00	0, (0,0)	0, (0,1)	0, (1,1)	0, (1,0)
1* → 01	0, (0,1)	0, (0,0)	0, (1,0)	0, (1,1)
1 → 11	0, (1,1)	0, (1,0)	0, (1,0)	1, (1,1)
2 → 10	0, (1,0)	0, (1,1)	1, (1,1)	1, (1,0)

4.2.2 Cell B digit coding

Using the same coding for cell B as was used in cell A gives $S_B(0) = S_A(0) \oplus C_A$, as required. However, the logic for $S_B(1)$ is not simple enough with this encoding. Swapping the output representations for 1 and 1* - that is replacing (0,1) by (1,1) and *vice versa*, did not impact the $S_B(0)$ logic, but allowed the $S_B(1)$ logic to be $S_B(1) = \neg S_A(1) \wedge S_A(0)$. Finally, $C_B = S_A(1) \wedge (\neg S_A(0) \vee C_A)$. The Karnaugh map for cell B is presented in Table 4.10.

Table 4.10 Karnaugh Map for Cell B

	0	**1**
0 → 00	0, (0,0)	0, (0,1)
1* → 01	0, (1,1)	0, (1,0)
1 → 11	0, (0,1)	1, (0,0)
2 → 10	1, (0,0)	1, (0,1)

4.2.3 Cell C digit coding

Finally, the output coding must match the input coding of cell A (i.e. 1* → (0,1) and 1 → (1,1)), and the input coding matches the output coding of cell B. By using the don't cares, the logic equations are $S_C(1) = \neg S_B(1) \wedge S_B(0) \vee C_B$, and $S_C(0) = S_B(0) \oplus C_B$. The final Karnaugh map of Cell C is presented in Table 4.11.

Table 4.11 Karnaugh Map for Cell C

	0	**1**
0 → 00	(0,0)	(1,1)
1 → 01	(1,1)	(1,0)
1* → 11	(0,1)	(1,0)
X → 10	(0,0)	(1,1)

Figure 4.4 shows the final CMOS gate implementation of the adder, where some further logic optimisation has been made (i) to cover the lack of AND and OR gates in CMOS, and (ii) to take advantage of CMOS complex gates. Note there is no FSEL input needed in this adder design.

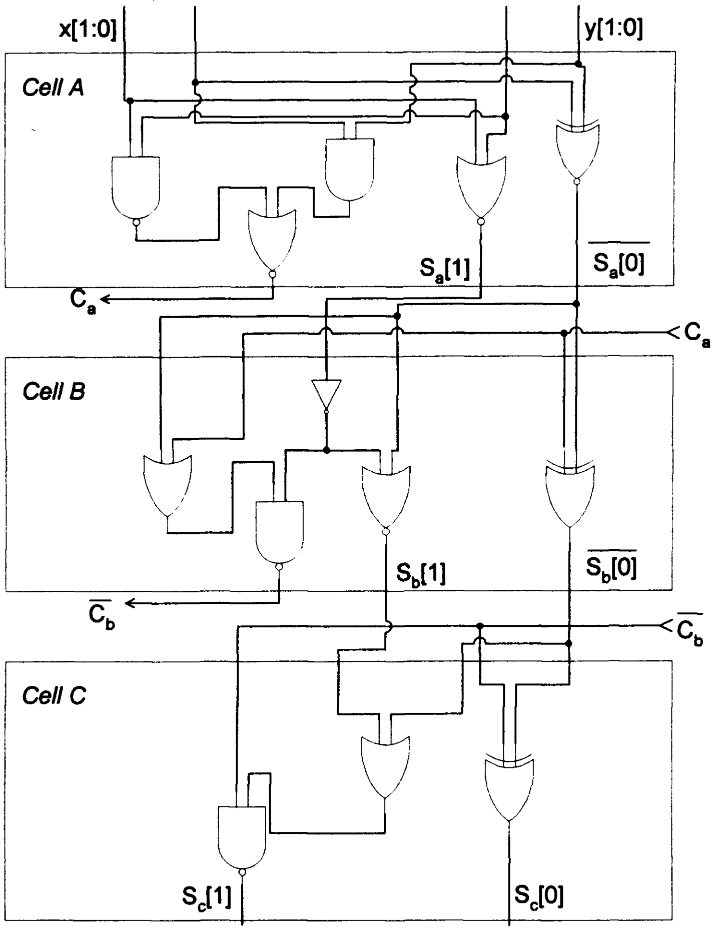


Figure 4.4 Overall gate implementation of new dual field (4:2) adder

For bit 0 of the adder, since the two carry inputs C_a and C_b are 0, $S_c[0]$ can be simplified to $x[0] \oplus y[0]$ and $S_c[1]$ is simplified to $\overline{Sa[1] + \overline{Sa[0]}}$. This ensures the data necessary for computing Montgomery modular multiple is ready as soon as possible. This also provides the opportunity to simplify the circuit design for the Montgomery modular multiple generation unit as will be seen later in section 5.3.

4.3 Unified field adders comparison

4.3.1 Area and Speed

The complete adder of Figure 4.4 was simulated using NC-Verilog and synthesised using Synopsis, which showed that the critical path (through the three XOR gates) was 1.50 ns using 0.18μm VLSI technology. By comparison, the four-input carry-save adders presented by Savaş in [121] are implemented as pairs of full adders with extra gates on the carry outputs to force carries to '0' (see Figure 3.10). Ignoring pipeline stages, Savaş' adder cells have a total CMOS logic gate count of 14 (counting XOR gates as two gates) as follows:

- 2 × 2 XOR (equivalent to 8 CMOS logic gates)
- 2 × 1 NOR
- 2 × 1 NOT
- 2 × 1 AOI CMOS complex gate

The proposed adder has a critical path length of only three XOR gates, with a CMOS logic gate count of only 13, made up as follows:

- 3 XOR/XNOR
- 2 NOR
- 1 NAND
- 1 NOT
- 2 OAI CMOS complex gates
- 1 AΘI CMOS complex gate

Logical Effort (see Appendix 2 for description of theory) can be used to assess the speed of the adder [131]. The reason why logical effort is used is because the results are close to reality and also it is technology independent. The delay is often represented in terms of FO4, which denotes "fanout of 4 inverters". This means the delay through an inverter that has to provide the output drive current sufficient to drive 4 other inverters of comparable sizes. Take the critical path of the (4:2) adder, which is the 3 XOR gates. These gates are connected to:

- XOR A: XOR, NOR, OR of OAI + wire
- XOR B: XOR, OR of OAI + wire
- XOR C: register

Assume the logical effort of an XOR is 4 as suggested in [131]. Also assume the logical effort of wire is 2/3 per fan-out. The input to XOR A is connected to an XOR and an AOI, such that the logical effort of the input is 4 (XOR) + 2 (AOI) + 2×2/3 for the wire, therefore the total logical effort of the input is 22/3. Table 4.12 shows the logical effort of the proposed adder.

Table 4.12 Logical effort of (4:2) unified field adder

	Logical effort *g*	**Branching effort *b***	**Electrical effort *h***	**Parasitic effort *p***	**Path effort *gb(h)***
Input	1	(22/3) / 4	1	-	22/12
XOR A	4	(4 + 5/3 + 6/3 + 3×2/3)/4	1	4	29/3
XOR B	4	(4 + 6/3 + 2×2/3)/4	1	4	22/3
XOR C	4	Assume 1 for register	1	4	4

The total path effort of the critical path for *w* word length is:

$F \quad = GBH = 22/12 \times 29/3 \times 22/3 \times 4$

$= 519.9$

The number of stages N needed including buffers can be calculated as follows:

$N \quad = \quad \text{Rnd}(\log_4 F)$

$= \quad \text{Rnd}(\log_4 519.9) = \text{Rnd}(\ln 519.9/\ln 4) = \text{Rnd}(4.51)$

$\text{Rnd } 4.51 = 5$

But XOR gates in CMOS have 2 stages. Therefore N is 6, not 5.

The stage load/drive α is calculated as follows:

$\alpha \quad = F^{1/N} = 519.9^{1/6}$

$= 2.84$

The delay along the critical path D is defined as $D = (N \times \alpha + P)/5$ in FO4 unit.

$D \quad = (6\times2.84 + 12)/5 = 5.80$ FO4

There are two paths through the unified field adder used by Savaş *et.al*: the first goes from inputs a or b to output S and the second goes from inputs a or b to output C_{out}. The logical effort of the path through two XOR gates to output S is shown in Table 4.13. The input is connected to an XOR and the AND of the AOI = 4 + 2 + 4/3 = 22/3. Table 4.13 shows the logical effort of Savaş' adder.

Table 4.13 Logical effort of path 1 of Savaş unified field adder

	Logical effort *g*	Branching effort *b*	Electrical effort *h*	Parasitic effort *p*	Path effort *gb(h)*
Input	1	(22/3) / 4	1	-	22/12
XOR A	4	(4 + 2 + 4/3)/4	1	4	22/3
XOR B	4	1	1	4	4

Path Effort of the critical path $F = GBH$ (critical path) = 22/12 × 22/3 × 4 = 53.8

The number of stages N needed including buffers can be calculated as follows:

N = Rnd ($\log_4 F$)
= Rnd (og_4 53.8) = Rnd (ln 53.8/ ln 4) = Rnd (2.87)
Rnd 2.87 = 3

As before, each XOR gate consists of two CMOS stages. Therefore, $N = 4$. The stage load/drive α and the delay D is calculated as follows:

$\alpha = F^{1/N} = 53.8^{1/4} = 2.71$
$D = (N \times \alpha + P)/5 = (4 \times 2.71 + 8)/5 = 3.77$ FO4

The second path goes through the first XOR gate, and then through the AOI and the NOR gates. The logical effort of this path is shown in Table 4.14.

Table 4.14 Logical effort of path 2 of Savas unified field adder

	Logical effort *g*	Branching effort *b*	Electrical effort *h*	Parasitic effort *p*	Path effort *gb(h)*
Input	1	(22/3) / 4	1	-	22/12
XOR A	4	(4 + 2 + 4/3)/2	1	4	44/3
AOI	2	(5/3 + 2/3) / (5/3)	1	4	14/5
NOR	5/3	1	1	2	5/3

Path Effort of the critical path $F = GBH$ (critical path) = 22/12 × 44/3 × 14/5 × 5/3 = 125.5

The number of stages N needed including buffers can be calculated as follows:

$$N \quad = \text{rnd}(\log_4 F)$$
$$= \log_4 125.5 = \ln 125.5 / \ln 4 = 3.49$$
$$\text{Rnd } 3.49 = 3$$

But, XOR gates consist of two CMOS stages. Therefore, $N = 4$. The stage load/drive α and the delay D is calculated as follows:

$$\alpha \quad = F^{1/N} = 125.5^{1/4} = 3.35$$
$$D \quad = (N \times \alpha + P)/5 = (4 \times 3.35 + 10)/5 = 4.68 \text{ FO4}$$

In terms of delay, the proposed design is 24% slower than Savas', however, the proposed design is capable of radix-4 operation, which will be beneficial to the implementation of the unified field multiplier. Also, the adder has one major advantage compared with Savaş's design and that is scalability.

4.3.2 Scalability

When compared with Savaş *et al.*'s design [121], shown in Figure 3.9, the unified multiplier presented here has the advantage that the Galois Field selection line does not cause extra delays due to potentially large fanouts. In Figure 3.9, the FSEL line has to drive $2w$ NOT gates and a long wire in the dual field adders, where w is the word-length of the adder.

The delay of the FSEL line driving $2w$ inverters can be estimated by using Logical Effort [131] as being roughly $\log_4 2w$ FO4 delays. However, this does not include load due to wire. Previously, $b = 2/3$ per fan-out was assigned to track delay. However, this is too small for this calculation because the figure of 2/3 assumed gates are placed next to each other. Here, gates are placed one adder apart so that b should increase to $5 \times 2/3 = 10/3$ because there are 5 gates in an adder. Therefore the delay of the buffer is estimated by $\log_4\{2\times1+10/3\}w = \log_4(16/3)w$. Hence, the ratio of delay due to transistor and wire to the delay due to transistors alone can be summerised as $\log_4(16/3)w$: $\log_4 2w$. Table 4.15 and Figure 4.5 show the relationship between delay due to transistor and wire to the delay due to transistors for different wordlength. The pipeline delay comprises this buffer delay and the adder delay, assuming that the partial product is generated in a prior pipeline stage.

Table 4.15 The ratio of delay due to transistor and wire to the delay due to transistors alone

wordlength	$\log_4(16/3)w$	$\log_4 2w$	Ratio $\log_4(16/3)w$: $\log_4 2w$
4	2.2075	1.5	1.4717:1
16	3.2075	2.5	1.2828:1
32	3.7075	3	1.2358:1
64	4.2075	3.5	1.2021:1
256	5.2075	4.5	1.1572:1

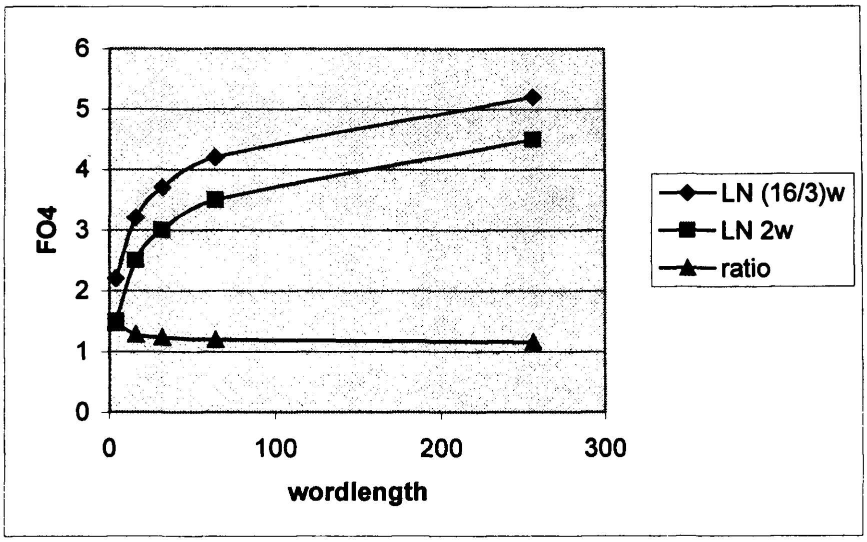


Figure 4.5 The ratio of delay due to transistor and wire to the delay due to transistors alone

In Figure 3.10, there are two critical paths through the adder: one starts with inputs *a* and *b* and traverses an XOR gate, a (2,2) AND-OR-invert (AOI) gate, and a NOR gate; the other starts with the FSEL line and comprises the FSEL buffer, an inverter, and the same NOR gate as the other path. The FSEL delay dominates the pipeline stage when the buffer delay becomes larger than the difference between these paths. From Logical Effort, the delay of the adder (path 2) was found to be 4.68 FO4. Now the delay of the inverter and NOR gate must be calculated.

Table 4.16 Logical effort of FSEL path in Savas unified field adder

	Logical effort *g*	Branching effort *b*	Electrical effort *h*	Parasitic effort *p*	Path effort *gb(h)*
Input	1	1	1	-	1
NOT	1	(5/3 + 2/3) / 5/3	1	1	7/5
NOR	5/3	1	1	2	5/3

Path Effort of the critical path $F = GBH$ (critical path) = 7/5 × 5/3 = 2.33

The number of stages N needed including buffers can be calculated as follows:

$$
\begin{aligned}
N \quad & = \text{Rnd}\,(\log_4 F) \\
& = \text{Rnd}\,(\log_4 2.33) = \text{Rnd}\,(\ln 2.33/\ln 4) = \text{Rnd}\,(0.61) \\
& \text{Rnd}\, 0.61 = 1 \\
\therefore N = 1 &
\end{aligned}
$$

Logical Effort says only one stage is needed. However, Savas's design has two stages, therefore $N = 2$. The stage load/drive α and the delay D is calculated as follows:

$$
\begin{aligned}
\alpha \quad & = 2.33^{\,1/N} = 2.33^{\,1/2} = 1.53 \\
D \quad & = (N \times \alpha + P)/5 = (2 \times 1.53 + 3)/5 = 1.21 \text{ FO4}
\end{aligned}
$$

Therefore when $\log_4(16/3)w > 4.68 - 1.21$, or $w = 23$, the FSEL buffer delay starts to dominate the critical path and affects the maximum clock rate achievable. Moreover, if only one bit is processed per pipeline stage, then this design could be vulnerable to Power Analysis cryptographic attacks as the word-length is small [1]. However, increasing the number of bits per stage increases the fan-out on the FSEL line, further degrading performance. Also, Savaş' adder is slower than the proposed unified adder when $w > 109$.

Table 4.17 (obtained from [121]) is a table to show the synthesis results of Savaş' adder using 1.2 μm CMOS technology. The delays include the time needed to form the partial product b_iA, the multiple of the modulus q_iM, and the necessary buffers. Ho *et.al* [132] proposed the idea that 1 FO4 = line-width (μm) × 360ps, for this instance, 1 FO4 = 1.2 × 360ps = 0.43ns. By using this equation the equivalent delay of Savaş adder in FO4 can be found as shown in Table 4.17.

Table 4.17 Savas' Adder results

Wordlength	Delay (ns)	Delay (FO4)	$\{b_iA$ + buf$\}$ + FSEL buf delay	Buffering FO4
16	6.87	15.9	15.9 – 4.68 = 11.22	11.22-5-1.5 = 4.72
32	9.22	21.4	21.4 – 4.68 = 16.72	16.72-5-1.5 = 10.22
64	12.55	29.2	29.2 – 4.68 = 24.52	24.52-5-1.5 = 18.02

The delay should increase logarithmically with wordlength, but the extra delay going from $w = 32$ to $w = 64$ causes extra increase in buffering delay

The last column calculates the delay due to buffering and forming b_iA or q_iM by subtracting the delay of the adder. Assuming b_iA (or q_iM) takes approximately 1.3 – 1.5 FO4 to be formed (using a 2-input NAND gate), and allowing 4 - 5 FO4 for register set-up and clk-to-q delays, the table shows that buffering dominates the delay for $w > 16$, see last column of Table 4.17. For example, at $w = 16$, 11.22 – 5 – 1.5 = 4.72 FO4 are needed for buffering, which is very close to the delay as calculated for the adder (4.68 FO4). This matches with the logical effort estimation.

The word length in the proposed design can be increased per pipeline stage as much as needed, without causing extra delay - so that this design is truly scalable. Moreover, this design could process more than one digit per pipeline stage without any extra delay due to field selection, although there would be additional delay due to the extra adders in each pipeline stage. However, the Multiplicand A and the Modulus M need to be converted into the novel redundant number coding, this can be done in parallel with the Multiplier B being fed to the row of partial product generators, thus avoiding any delay due solely to field selection.

The next chapter shows how a complete dual-field multiplier can be designed based on the dual-field adder introduced here.

5 Unified Field Multiplier

In this chapter, the design of the proposed unified field multiplier will be described. The definition of unified field architecture is one architecture that is able to perform operations in both prime field GF(p) and binary extension field GF(2^n) using the same data path. The proposed multiplier has the following properties:

- Scalable - the hardware structure can be scaled up or down by reusing, replicating or truncating, without affecting clock period

- Fast – the performance of the multiplier should not be compromised by being dual-field instead of dedicated to single field

- Impartial – the multiplier must not favour one prime number or an irreducible polynomial over others for flexibility of applications

Unified field multipliers have the advantages of low manufacturing cost, they also provide compatibility and flexibility by being interoperable. Even though GF(p) and GF(2^n) have very different properties, their representations and structures are very similar. They can both be represented as bit strings and their arithmetic structures are the same except that GF(p) performs modulo a prime p and GF(2^n) performs modulo an irreducible polynomial $M(x)$. This provides the opportunity to implement arithmetic unit that is interoperable between either fields.

Chapter 4 described the unified field redundant adder that is employed in the proposed unified field multiplier. The technique of embedding field information into the encoding of the data was introduced in section 4.2. The same encoding will be used throughout the implementation of the multiplier.

In the first part of this chapter, different common modular reduction techniques are discussed, which includes the chosen Montgomery modular multiplication. The second section shows the implementation of the proposed multiplier design and the final section of this chapter discusses the strengths of the proposed design compared with the previously proposed architectures.

5.1 Modular Multiplication Algorithm

Modular multiplication is required for many cryptosystems such as RSA and ECC because it allows encrypted data, which are very large in size, to be securely stored in public domain but could only be decrypted by the users who hold the authorised key. Modular multiplication means that, for A, B and $M \leq n$-bit:

$$R = A \cdot B \bmod M \tag{5.1}$$

Modular multiplication is computed using the multiply-and-reduce method, which can be expressed by the following equations:

$$\text{Multiply}: X = A \times B \tag{5.2}$$

$$\text{Reduce}: R = X \bmod M \tag{5.3}$$

The multiplication part is relatively simple to compute, except that the numbers involved tend to be very big and the size of partial product result X becomes $2n$-bit, this should be reduced by the modular reduction operation to n-bit. Modular reduction is the remainder R of a division such that:

$$R = X - \left\lfloor \frac{X}{M} \right\rfloor M \tag{5.4}$$

$$R = X - qM \tag{5.5}$$

The modulus M and the partial product result X of base b are defined as follows:

$$M = \sum_{i=0}^{n-1} m_i b^i \,, \qquad 0 < m_{n-1} < b \text{ and } 0 \le m_i < b \text{, for } i = 0, 1, \ldots, n\text{-}2 \tag{5.6}$$

$$X = \sum_{i=0}^{l-1} x_i b^i \,, \qquad 0 < x_{l-1} < b \text{ and } 0 \le x_i < b \text{, for } i = 0, 1, \ldots, l\text{-}2 \tag{5.7}$$

The "pencil-and-paper" division approach for modular reduction requires n subtractions and shifts, one long multiplication and one final subtraction. Knuth [133] formalized the "pencil-and-paper" method to give the so-called classical algorithm. The pseudo code of the classical method is shown in Figure 5.1:

```
{Pre-condition: $0 \le A \times B < M \times r^n$}
$R := A \times B$
For $i := n-1$ down to 0
Do
    Begin
        $q := R$ div $(M \times r^i)$;
        $R := R - q \times M \times r^i$;
        { Invariant: $0 \le R < M \times r^i$ & $R \equiv (A \times B) \bmod M$ }
    End ;
{Post-Condition : $R = (A \times B) \bmod M$}
```

Figure 5.1 The classified pen-and-paper division method

Since the quotient q is not required in modular reduction, working out the exact quotient is extremely time consuming, and so different methods have been introduced to speed up the process, such as by estimating the quotient. Knuth provided methods to estimate the quotient based on the fact that the condition $X/M < b$ (b is the base of the number) is equivalent to $\lfloor X/b \rfloor < M$. Since $R = X - qM$ and q is an integer such that $0 \le R < M$ therefore an approximation of q, denoted by $\hat{q}$, can be obtained by dividing the most significant digits of X, x_{l-1} and x_{l-2}, where l is the wordlength of X, by the most significant digit of M, m_{k-1}. If the result is b or larger, then replace it by b-1 such that:

$$\hat{q} = \min(\left\lfloor \frac{x_{l-1}b + x_{l-2}}{m_{k-1}} \right\rfloor, b-1) \tag{5.8}$$

The pseudo code of the Knuth algorithm is given in [134] and is shown in Figure 5.2.

```
if ($X > Mb^{l-n}$) then
        $X = X - Mb^{l-n}$;
for ($i = l$-1; $i > n$-1; $i$--) do {
        if ($x_i == m_{n-1}$) then
                $q = b$-1;
        else
                $q = (x_i b + x_{i-1})$ div $m_{n-1}$;
        while ($(q(m_{n-1}b + m_{n-2}) > x_i b^2 + x_{i-1}b + x_{i-2})$ do
                $q = q$-1
        $X = X - qMb^{i-n}$;
        if ($X < 0$) then
                $X = X + Mb^{i-n}$;
        }
```

Figure 5.2 Knuth Algorithm ($m_{n-1} \geq \lfloor b/2 \rfloor$)

Dhem [135] suggested that this method is more advantageous in the case where a fast and large divider is readily available; otherwise a hardware divider is slower and more expensive than a hardware multiplier.

Another example of a method that improves the speed of modular multiplication by estimating the quotient q is called Brickell method [136]. This method makes use of delayed carry adders, which is a modified version of carry-save adders. This method determines the multiple of modulus M to be subtracted from the partial multiplication as a result of assessing the top digits of the partial multiplication results. This is similar to SRT division.

In general, the three most common methods to compute modular multiplications are:

- Interleaved modular multiplication – compute a multiplication followed by a reduction. This approach makes use of the usual modular multiplication order, which multiplies from the most significant bit to the least significant bit. This method has the benefit of keeping the register requirement of the partial sum to n-bits and thereby saving register space.

- Barrett modular multiplication [137] [138] – The pseudo code of Barrett's algorithm is shown in Figure 5.3 [134]. Barrett suggested pre-computing the inverse of each modulus M at the beginning of computation:

  $$W = \frac{b^{2n}}{M} \tag{5.9}$$

  Where n is the wordlength of the operands. Instead of division by M, multiplication of W which has $n + 1$ digits, is performed because division is less efficient. Typically, division is 10 times or more slower than multiplication on a microprocessor.

  $$R = X - (X \cdot W) \cdot M \tag{5.10}$$

  This method approximates the quotient by using a scaled estimate of the modulus' reciprocal such that:

  $$\hat{q} = (\frac{X}{b^{2n-1}} \cdot W) \tag{5.11}$$

  This means multiplying the most significant (n+1) digits of X by W, which is the inverse of M. The n most significant digits of the approximation of the quotient is then multiplied by M. The estimated remainder is attained by subtracting the n+1 least significant digits of $\hat{q}M$ from the corresponding part of the partial product X. This can be summarised as follows:

$$\hat{R} = (X \bmod b^{n+1} - (\hat{q}M) \bmod b^{n+1}) \bmod b^{n+1} \tag{5.12}$$

Bosselaers [134] explained that, in the calculation of the product $\hat{q} = (\frac{x}{b^{2n-t}} \cdot W)$, the calculation of the t-2 least significant digits can be avoided because the carry from position t to position t+1 can be accurately estimated by calculating the digits at position t-1 and t. This means that the estimation of the remainder is similar to that of the quotient, such that only a partial multiplication is needed. Dhem [135] provided an improved Barrett's algorithm by introducing a new parameter α that would refine the estimation on the quotient. Another example of implementation that makes use of the Barrett's algorithm can be found in [139].

```
$q$ = (($x$ div $b^{n-1}$)$\mu$) div $b^{n+1}$;
$x = x$ mod $b^{n+1}$ – ($qm$) mod $b^{n+1}$;
if ($x < 0$) then
        $x = x + b^{n+1}$;
while ( $x \geq m$) do
$x = x$ - $m$
```

Figure 5.3 Barrett's Algorithm ($\mu = b^{2n}$ div m)

- Montgomery modular multiplication [118] – The main purpose of this algorithm is to find an appropriate multiple of the modulus to be added to partial product $A \times B$ so that the lowest k bits will become 0. If the lowest k bits are 0, instead of the need for modular division, k bit right shift is performed.

Unlike usual multiplication practice, Montgomery modular multiplication chooses digits from least to most significant bit, and shifts down during each iteration. The modulo multiple q is computed from the lowest digits of the modulus M and partial sum (PS) where $PS = A \times b_i + R$ and R is the result from the previous iteration. The advantage of this design is that there is no need to wait for any carry propagation. The Montgomery's algorithm is described in Figure 5.4. Let R be the remainder, r be the radix, M is the modulus involved and gcd(M, R) = 1. The quotient q is chosen such that $R+qM$ is a multiple of r. Many implementation of the Montgomery modular multiplication can be found, such as reference [156] and reference [164], it will be described in more detail later on in this chapter.

```
{Pre-condition: $0 \le A \le r^n$}
   R:= 0;
   For $i := 0$ to $n-1$ do
   Begin
        $q_i := (-(R_0 + a_0 b_i) m_o^{-1}) \bmod r$;
        $R := (R + A \times b_i + q \times M) \operatorname{div} r$;
        {Invariant: $0 \le R < M+B$}
   End;
        {Post-condition: $R \equiv (A \times B \times r^{-n}) \bmod M$}
```

Figure 5.4 Montgomery's Algorithm

These three common modular reduction algorithms – the classical method, the Barrett's method and the Montgomery reduction – were compared and the following results were found [134]:

Table 5.1 Complexity of the three reduction algorithm in reducing a 2*k*-digit number *x* modulo a *k*-digit modulus *m*

Algorithm	Modulo Multiplication			Ordinary Multiplication
	Classical	Barrett	Montgomery	
Multiplications	$n(k+2.5)$	$n(n+4)$	$n(n+1)$	n^2
Divisions	N	0	0	0
Precalculation	Normalization	b^{2n} div m	$-m_0^{-1}$ div b	None
Argument Transformation	None	None	m-residue	None
Postcalculation	Unnormalization	None	Reduction	None
Restrictions	None	$x < b^{2k}$	$x < mb^k$	None

Table 5.2 Execution times for the reduction of a 2*k*-digit number modulo a *k*-digit modulus *m* for the three reduction algorithms compared to the execution time of a *k* x *k* - digit multiplication (*r* = 2 16, on a 33 MHz 80386 based PC with WATCOM C/ 386 9.0)

K	Length of *m* in bits	Times in mseconds			
		Classical	Barrett	Montgomery	Multiplication
8	128	0.278	0.312	0.205	0.182
16	256	0.870	0.871	0.668	0.632
32	512	3.05	2.84	2.43	2.36
48	768	6.56	5.96	5.33	5.19
64	1024	11.39	10.23	9.33	9.12

The most time consuming operations within the three algorithms are the multiplications and divisions, therefore comparing the number of multiplications and divisions needed should provide a fairly accurate indication of the comparisons between the three algorithms. Note that Table 5.1 shows only the number of multiplications and divisions for the reduction operation. If only the reduction operation is considered, assuming that the arguments are twice the length of the modulus, Table 5.2 shows that Montgomery's algorithm is faster than the other two algorithms. Note that Montgomery's algorithm is only applicable when the modulus *m* is gcd $(m, r) = 1$. For the purpose of this thesis, *m* is assumed to be an odd number.

Table 5.2 shows the execution time taken to perform the reduction operation on a 2*k*-digit number modulo a *k*-digit modulus *M* where the radix *r* is 2^{16} using the three different algorithms. This is performed on a 33 MHz based PC with 32-bit compiler WATCOM C/386 9.0. The timing shown in Table 5.2 confirms the assumptions used in Table 5.1. However, due to the pre- and post-calculations and the *m*-residue transformation required by Montgomery's algorithm, the benefits of the high reduction speed of Montgomery's algorithm is maximised in the case where the numbers involved in the modular reduction are very big, such as in the case of modular exponentiation for cryptography.

In general, Montgomery multiplication is the chosen method for modular multiplication because the division operation required for modulo reductions is replaced by shift operations, which is particularly beneficial for implementation. It has been shown that Montgomery's algorithm is efficient especially when the multiplication calculation is intensive, which is often the case in cryptography. Other advantages of Montgomery multiplication are [140]:

- Scalable
- Highly parallel
- Suitable for pipelining
- Use only addition instead of subtraction

Section 5.2.1 describes the implementation of Montgomery multiplication in *GF*(*p*) and section 5.2.2 shows the implementation of Montgomery multiplication in $GF(2^n)$. In section 5.2.3, the means to implement a unified field multiplier will be explained. It shows that even though the two fields are different in nature, dual field arithmetic hardware can be easily implemented because they are structurally very similar.

5.2 Unified Field Montgomery Multiplication

5.2.1 Montgomery Multiplication in GF(*p*)

There have been many proposed designs on Montgomery Multiplication in *GF*(*p*), such as [141], [142] and [143].

The elements of *GF*(*p*) are made up of integers {0, 1, 2, …, *p*-1} and radix $r = 2^k$. They are represented as:

$$A = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)_{2^k}$$

$$B = (b_{n-1}, b_{n-2}, \ldots, b_1, b_0)_{2^k}$$

$$R = (r_{n-1}, r_{n-2}, \ldots, r_1, r_0)_{2^k}$$

Addition and Multiplication operations in GF(p) are performed as regular integer addition or multiplication, therefore carry propagation are involved. The addition or multiplication result will then be reduced by the modulus so that the final result will be smaller than the modulus.

The Montgomery Multiplication algorithm in *GF*(p) is shown in Figure 5.5.

```
Input: $A$, $B$, $M$ $(A < M, B \geq 0)$
Output: $R = AB\, 2^{-n} \bmod p$

$R := 0$
for $i = 0$ to $n$-1
        $q_i := (s_0 + a_0 b_i)(-m_o^{-1}) \bmod r$
        $S := (S + A \times b_i + q_i \times M) div r$
if ($S > M$) then $S = S - M$
```

Figure 5.5 Montgomery Multiplication in GF(p)

5.2.2 Montgomery Multiplication in GF(2^n)

For *GF*(2^n), elements are represented in polynomials of degree ≤ n-1 and the coefficient ∊ *GF*(2) if polynomial basis is used:

$$A(x) = a_{n-1}x^{n-1}, a_{n-2}x^{n-2}, \ldots, a_1 x, a_0$$

$$B(x) = b_{n-1}x^{n-1}, b_{n-2}x^{n-2}, \ldots, b_1 x, b_0$$

The irreducible polynomial of degree m is represented as follows:

$$M(x) = x^n + m_{n-1}x^{n-1}, m_{n-2}x^{n-2}, \ldots, m_1 x, m_0$$

Unlike addition in *GF*(*p*) which involves carry propagation, no carry propagation is required for addition in *GF*(2^n), so the degree of the resulting polynomial will not exceed degree *n*, this means that the final reduction step required in *GF*(*p*) multiplication is not necessary here. Therefore bit-wise modulo-2 addition is used instead of normal addition with carry, thus XOR gates are utilised.

The Montgomery Multiplication algorithm in *GF*(2^n) is shown in Figure 5.6.

```
Input: $A(x)$, $B(x)$, $M(x)$
Output: $R(x)$

$R(x) := 0$
for $i = 0$ to $n$-1
    $q_i(x) := (s_0(x) + a_0(x)b_i(x))(-m_0^{-1}(x)) \bmod x^k$
    $S(x) := (S(x) + A(x) \times b_i(x) + q_i(x) \times M(x)) div x^k$
```

Figure 5.6 Montgomery multiplication in GF(2^n)

Some examples of previously proposed Montgomery multiplier can be found in [144], [145].

5.2.3 Unified Field Montgomery Multiplication

As presented in section 4.2 and 4.3, it can be seen that the elements of the two fields can be presented using almost the same data structures. For example for *GF*(7) with modulus = 7, the elements can be represented as:

GF(7) = {000, 001, 010, 011, 100, 101, 110}

For *GF*(2^3) with irreducible polynomial = x^3+x+1, the elements can be represented as:

GF(2^3) = {000, 001, 010, 011, 100, 101, 110, 111}

Also, the structures of the algorithm for basic arithmetic operations in both fields are very similar. The main difference is that arithmetic operations in *GF*(*p*) are carried out like regular arithmetic operations, where carry propagation is involved. Whereas for *GF*(2^n) operations, bit-wise modulo-2 operation is performed. This provides the possibility of implementing unified multiplier architectures. Some previously proposed unified field multipliers could be found in [121], [123], [124] and [122].

The remaining sections in this chapter will describe the proposed unified field multiplier. This design is in word-serial nature to show the proposed method to implement dual field multiplier without the need to have a field-select signal. This solution has a very high fan-out because it is fed into multiple gates, as in previously published designs.

Section 5.3 will describe the overall structure design of the multiplier, implementation details of individual module involved will be shown in subsequent sections.

5.3 Proposed Word-Serial Montgomery Multiplier Architecture

Figure 5.8 shows the proposed word-digit dual-field multiplier architecture, which corresponds to the step-by-step codes shown in Figure 5.7. It involves two partial product generations, two partial product summations and one modulo multiple determination process.

```
Input: $A, B \in \mathrm{GF}(p)$ and $n = \lceil \log_2 M \rceil$
Output: $R \in \mathrm{GF}(p)$

$R := 0$
for $i = 0$ to $n$-1
        $PP := Ab_i$
        $PS := R + PP$
        $R := PS + \mathrm{qM}$
        $R := R/2$
if $\mathrm{R} \geq \mathrm{m}$, then $\mathrm{R} := \mathrm{R}\text{-}\mathrm{m}$
return R
```

Figure 5.7 Bit-wise Montgomery Multiplication (step-by-step)

Because of the novel coding system used, binary numbers are encoded into redundant numbers. Details on the coding systems are explained earlier in section 5.2, where the description of implementation of proposed redundant adder can also be found. Section 5.3.1 shows the implementation of the partial product generator and the binary to redundant encoder. The implementation of the circuit for modular reduction is found in section 5.3.2.

The overall structure of this multiplier is shown in Figure 5.8. It shows that the multiplier comprises six different modules: (1) Binary to Redundant number encoder; (2) Partial Product Generator; (3) (4:2) adders for partial products summation; (4) Modulus Multiplier Digit Selection; (5) Modulus Multiple Generator; (6) (4:2) adders for modulo reduction. However, only four different modules are required because the two (4:2) adders required are the same, as are the partial product and modulus multiple generators. The (4:2) adder has already been introduced, so the following sections shall present the design of binary to redundant number encoder and also the design of the partial product generator. The modular reduction will be presented in the last section of the Chapter.

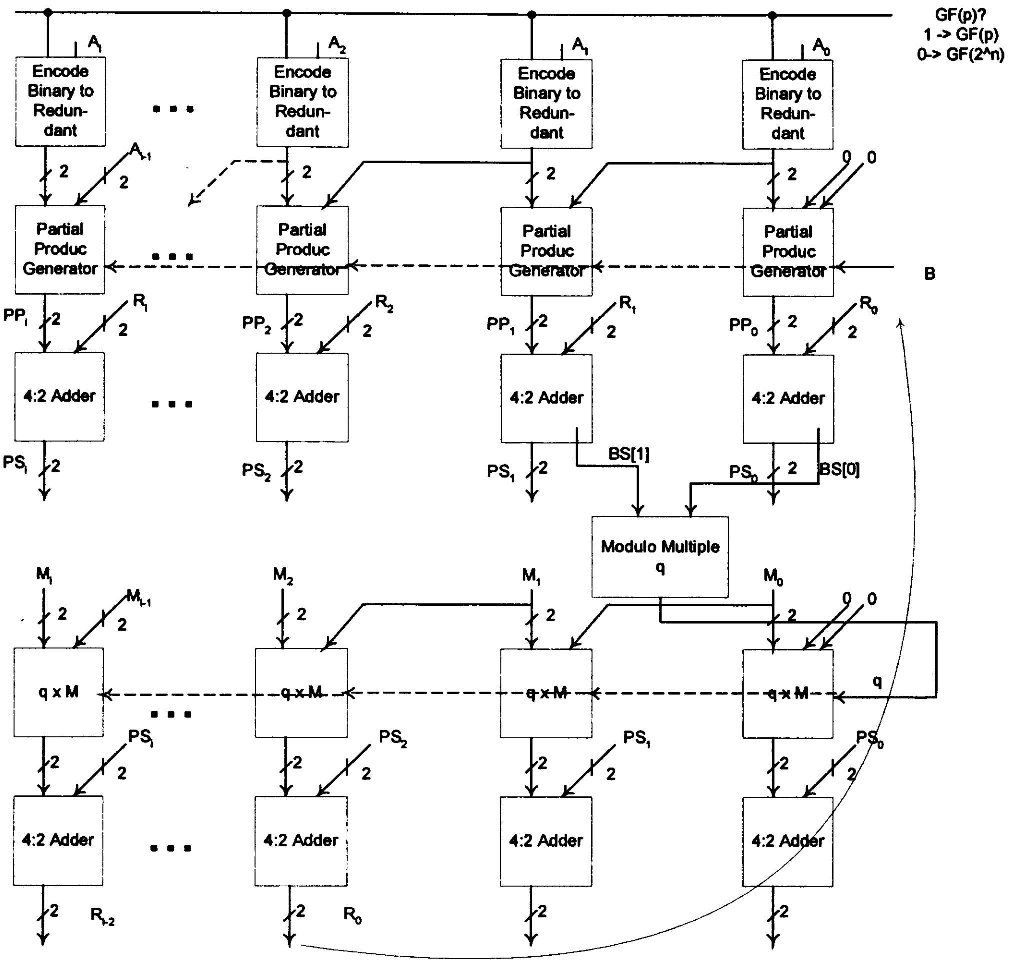


Figure 5.8 Proposed Word -Digit Dual-Field Multiplier Architecture

5.3.1 Unified radix-4 Partial Product Generator

5.3.1.1 Radix-2 integer multiplication

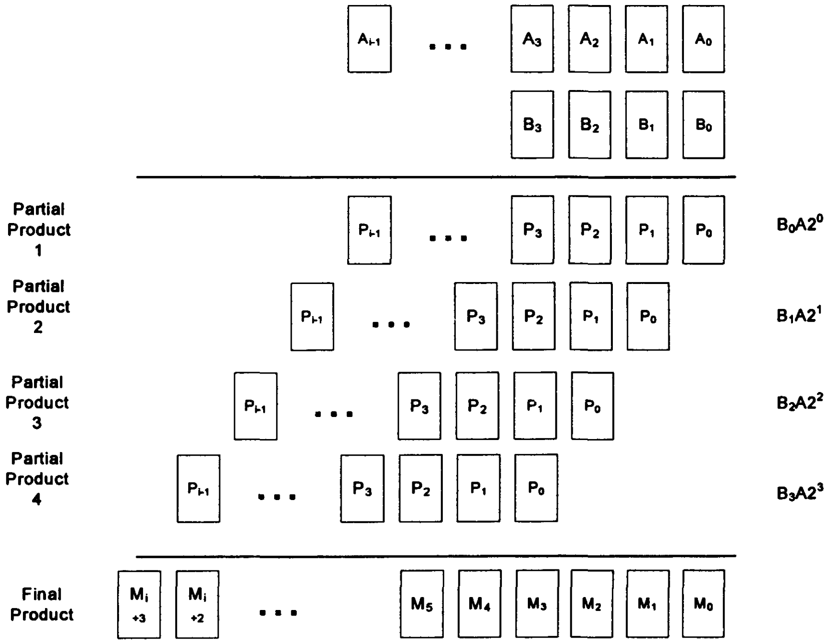
The multiplication of unsigned radix-2 $A \times B = P$ multiplication is discussed in this section. A, B and C denote the multiplier, the multiplicand and the product, they are represented as follows:

Multiplier A: $A_{i-1}2^{i-1} + A_{i-2}2^{1-2} + \ldots + A_1 2^1 + A_0 2^0$

Multiplicand B: $B_{i-1}2^{i-1} + B_{i-2}2^{1-2} + \ldots + B_1 2^1 + B_0 2^0$

Product $(A\times B) = P$: $P_i 2^{2i-2} + P_{i-1}2^{2i-3} + P_{i-2}2^{2i-4} + \ldots + P_1 2^1 + P_0 2^0$

Figure 5.9 shows the multiplication of an i-bit number by a 4-bit number. The product is formed by summing all partial products, each partial product is produced by $B_i{\cdot}A{\cdot}2^i$. Since B_i is in $\{0,1\}$, each partial product term can only be equal 0 or $A{\cdot}2^i$. This operation can be represented by the logical AND. Therefore, the binary multiplication is equivalent to the summation of partial products, which is either 0 or shifted version of A. Avizienis gives an example of a radix-2 multiplier implementation [128].



A_{i-1} ... A_3 A_2 A_1 A_0
B_3 B_2 B_1 B_0
Partial Product 1
P_{i-1} ... P_3 P_2 P_1 P_0
B_0A2^0
Partial Product 2
P_{i-1} ... P_3 P_2 P_1 P_0
B_1A2^1
Partial Product 3
P_{i-1} ... P_3 P_2 P_1 P_0
B_2A2^2
Partial Product 4
P_{i-1} ... P_3 P_2 P_1 P_0
B_3A2^3
Final Product
M_{i+3} M_{i+2} ... M_5 M_4 M_3 M_2 M_1 M_0

Figure 5.9 unsigned Radix-2 AxB Multiplication

5.3.1.2 Radix-4 multiplication

By increasing the radix of the multiplier to 2^k, the number of iteration for partial product calculation is reduced because this is equivalent to operating k bits of the radix-2 multiplier per iteration.

The implementation of high radix multiplier is becoming more common: for example, designs such as [146], [147], [148] and [149] all describe high-radix modular multiplication implementations. Walter [150] discussed the trade off between time and space when implementation high-radix design. It is said that by having a moderate increase in radix value, it can provide a faster alternative to that of the radix-2 designs.

Figure 5.10 shows the multiplication of an *i*-bit number by a 4-bit number in radix-4. In radix-2 multiplication, the partial products are either 0 or shifted version of *A*. In radix-4 multiplication, one needs to consider the multiples: 0×*A*, 1×*A*, 2×*A* and 3×*A*. Figure 5.11 depicts the partial product generation in radix-4 to compute the multiples: 0×*A*, 1×*A*, 2×*A* and 3×*A*. Note that in the diagram the multiple 3×*A* is assumed to be pre-computed.

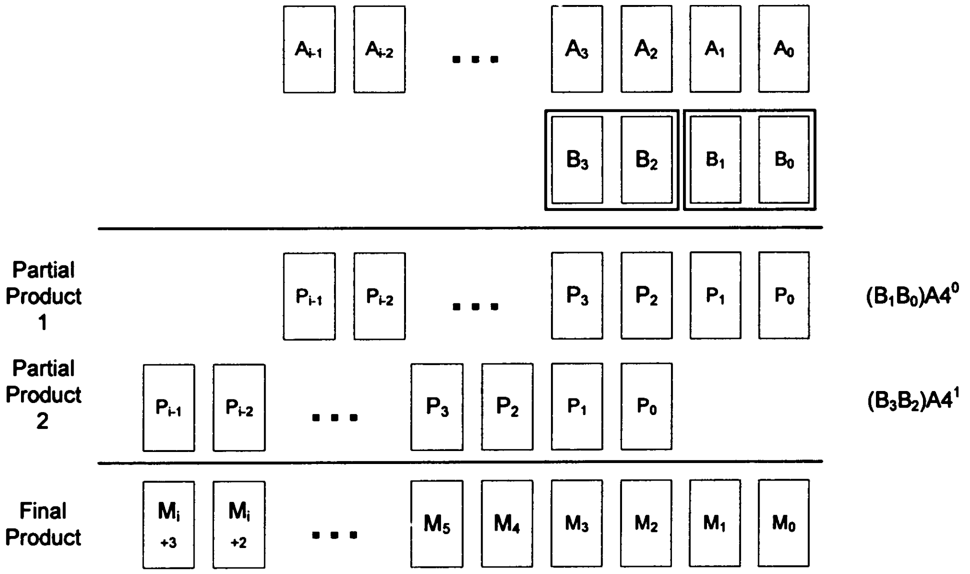


Figure 5.10 Radix-4 multiplication

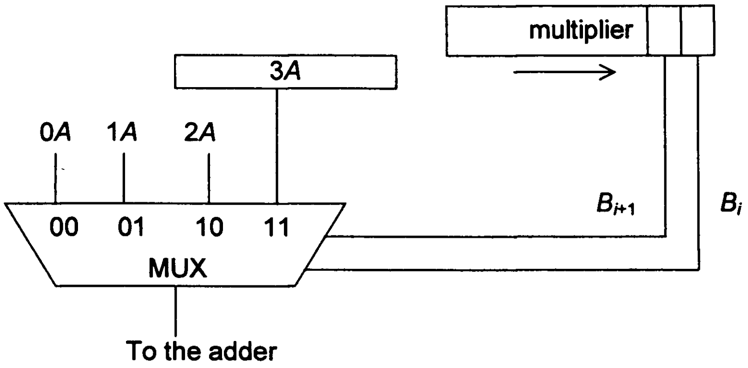


Figure 5.11 partial product generation in radix-4 with pre-computation of 3x*A*

Groβschädl [151] described a unified radix-4 partial product generator. For GF(*p*), the generation of partial products is performed according to the modified Booth recoding technique [152] (see Figure 5.12). For GF(2^n), the partial products are generated in the same way as this is done by a digit-serial polynomial-multiplier with a digit size of $d = 2$. Therefore, two bits of the multiplier are processed in either case, generating one partial product. The unified partial product generator is shown in Figure 5.13.

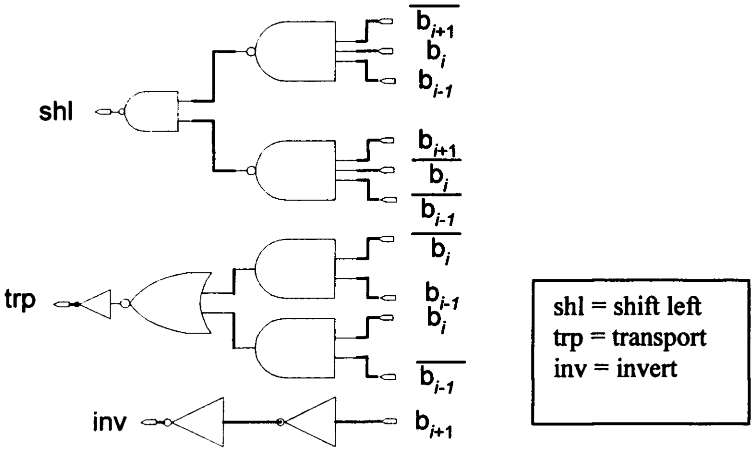


Figure 5.12 Groβschadl Booth encoder circuit

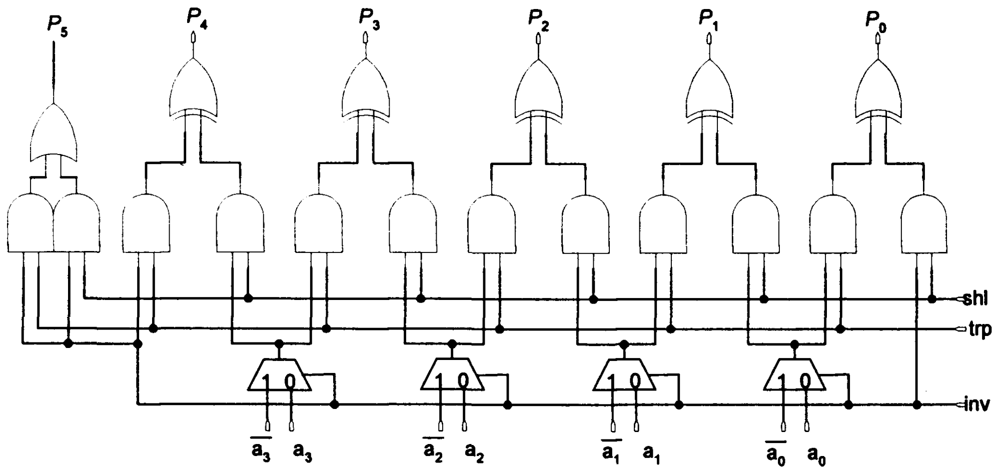


Figure 5.13 Groβschadl unified radix-4 partial product generator

In the proposed design, the two digits input to the (4:2) adders - namely, the result of the previous iteration, R_n, and the partial product, PP_n, both have the digit set, $d \in \{0, 1, 1^*, 2\}$. The partial product generation is decomposed into two steps: firstly, the selected Galois Field is embedded into the multiplicand word by encoding it using the novel $d = 1^*$ representation; secondly, the radix-4 partial product is derived by using the available redundant $d = 2$ representation. The first of these steps, embedding the Galois Field, is implemented by the simple circuit shown in Figure 5.14. This is the same coding system as used for the adder described in Chapter 4. Even though the GF(p) line may have a very high fan out when the wordlength is long (see Figure 5.8), buffers can be added to the signal; moreover, the GF(p) line is out of the critical path, hence the buffering does not affect the overall delay of the multiplier.

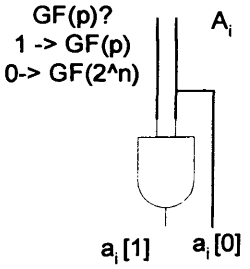


Figure 5.14 Field-Embedded Binary Number Encoder

Every two bits of the multiplier word, B, are recoded as a radix-4 digit, and the multiplicand, A, then multiplied by the recoded bit to yield the appropriate partial product, as shown in Table 5.3. Figure 5.15 shows the logic diagram of the partial product generator, including the Field-embedded binary number encoder, which is similar to that shown in Figure 5.11. It is seen to be simpler than the standard radix-4 Booth's encoder, such as the one shown in Figure 5.13 [151]. In particular, the negative multiple increment bits that occur in Booth's coding are avoided, as these can increase the logic depth of the adder array and negative partial products do not exist in modular arithmetic. Note how the availability of the redundant digit, $d = 2$, at the (4:2) adder input means there is no need for a carry-propagate addition when encoding the radix-4 digit of 3.

Table 5.3 Radix-4 Partial Product Generation

(B_i, B_{i-1})	**Radix-4 digit**	**Partial product, PP_i[1:0]**
00	0	0
01	1	A
10	2	Left shift A 1 bit
11	3	$1A + 2A$

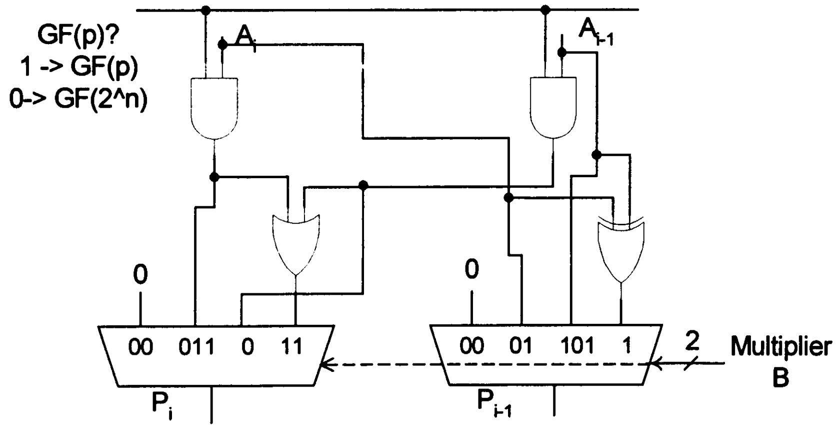


Figure 5.15 Radix-4 Partial Product Generator

5.3.2 Unified Modulo Reduction

In order to carry out Montgomery reduction, one needs to work out the multiple of the modulus such that when added to the partial product, the result of the last two digits (for radix-4) becomes zero. Figure 5.8 shows that the modular multiple selection (i.e. determining q_i) causes irregularity in the design and is on the critical path. Therefore, effort is needed to reduce the delay by taking into consideration pre-known factors as early on in the calculation as possible. For example, the modulus M is always an odd number (because $r = 2^n$), so that the last bit of M, $M[0]$, will always be 1. Therefore the information presented in Table 5.4 regarding the two LSB's of $q_i \cdot M$ is already known before any modulo reductions are performed.

Table 5.4 Multiple of *M*

Multiple of M	$M[1,0] = 01$	$M[1,0] = 11$
1	01	11
2	10	10
3	11	01

Table 5.5 shows what value of q_i is required to ensure $R = BS[1, 0] + q_i \cdot M = 00$ as a function of the selected Galois Field, where the two LSB's of the Partial Sum, denoted by $BS[1:0]$, are in conventional binary form rather than in redundant form.

Table 5.5 Selection of Modulo Multiple, $q_i \times M$

	Partial Binary Sum, BS[1:0]	$q_i[1:0]$ if M[1,0] = 01	$q_i[1:0]$ if M[1,0] = 11
GF(p), GF = 1	00	00	00
	01	11	01
	10	10	10
	11	01	11
GF(2^n), GF = 0	00	00	00
	01	01	11
	10	10	10
	11	11	01

From Table 5.5, it is easy to see that $q_i[0] = BS[0]$ independently of both the Galois Field and $M[1:0]$. However, $q_i[1]$ is a function of $M[1]$, $BS[1:0]$, and the Galois Field flag, *GF*. Figure 5.16 shows a simple circuit implementing the necessary logic organised as a multiplexer controlled by $BS[0]$.

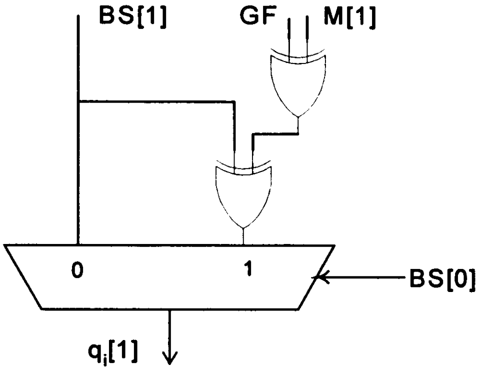


Figure 5.16 $q_i[1]$ logic

Montgomery's modular reduction technique is performed on non-redundant binary numbers. Therefore, the redundant representation returned by the (4:2) adders must be converted to binary to obtain the bits $BS[1:0]$. Table 5.6 presents this conversion process, where PS_i denotes the two bits representing the partial sum at bit position i (see Figure 5.8). Note that $PS_1[1]$ is not included in the Table, because it is weighted +2 and so has no effect on the value of $BS[1]$.

The Table shows that $BS[0] = PS_0[0]$. In fact, since C_a and C_b to the (4:2) adder are both 0, $BS[0] = PP_0[0] \oplus R_0[0]$, and is available much earlier than $BS[1]$. The logic for $BS[1]$ is presented in Figure 5.17 as a multiplexer controlled by $BS[0]$, in common with Figure 5.16. Merging Figure 5.16 and Figure 5.17 yields the simplified circuit for $q_i[1]$ shown in Figure 5.18.

Table 5.6 Binary Conversion

$PS_1[1]$	$PS_1[0]$	$PS_0[1]$	$PS_0[0]$	digit[1]	digit[0]	BS [1]	BS [0]
0	0	0	0	0	0	0	0
0	0	0	1	0	1*	0	1
0	0	1	0	0	2	1	0
0	0	1	1	0	1	0	1
0	1	0	0	1*	0	1	0
0	1	0	1	1*	1*	1	1
0	1	1	0	1*	2	0(x)	0(x)
0	1	1	1	1*	1	1(x)	1(x)
1	0	0	0	2	0	0	0
1	0	0	1	2	1*	0(x)	1(x)
1	0	1	0	2	2	1	0
1	0	1	1	2	1	0	1
1	1	0	0	1	0	1	0
1	1	0	1	1	1*	1(x)	1(x)
1	1	1	0	1	2	0	0
1	1	1	1	1	1	1	1

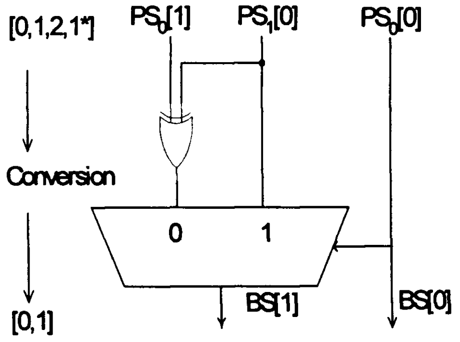


Figure 5.17 Logic for *BS*[1]

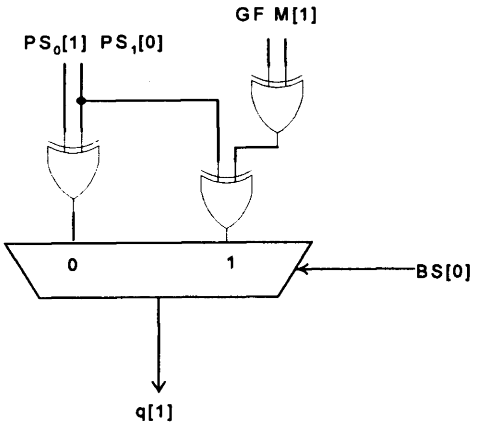


Figure 5.18 Simplified logic for $q_i[1]$ combined with *BS*[1] logic

Once q_i has been determined, the modulo M is multiplied by q_i using the modulo multiple generator shown in Figure 5.19, which has the same logic design as the partial product generator presented earlier. Finally, the multiple, $q_i{\cdot}M$, is then added to partial sum (*PS*) using the same modified (4:2) adder as shown in Figure 4.4. Note that in Figure 5.8, the least significant four bits (2 binary bits) are discarded as they are now zero and what was R_2 is now fed back to the partial product adder as R_0.

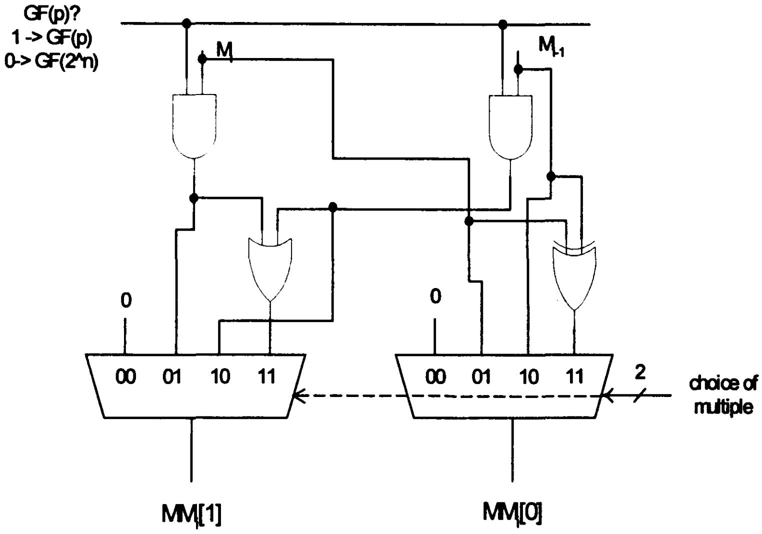


Figure 5.19 Modulo multiple generator

5.3.3 Carry absorption Unit

In order to perform Montgomery modulo reduction, the last radix-4 digit of the partial result was forced to zero by adding an appropriate multiple of the modulus. However, due to the design of the specially adapted (4:2) redundant adder, the output of the last two bits may not necessarily both become 0; instead they may stay in their redundant form, i.e. 2. Therefore, effort has to be made to ensure that a ‘2’ digit will be changed to 0 plus a carry. This can be easily done with an AND gate as shown in Figure 5.20. Since the carry inputs to the first of the (4:2) adders are not used (and were previously set to zero) the carry caused by changing a 2 to a 0 can be absorbed by that carry input.

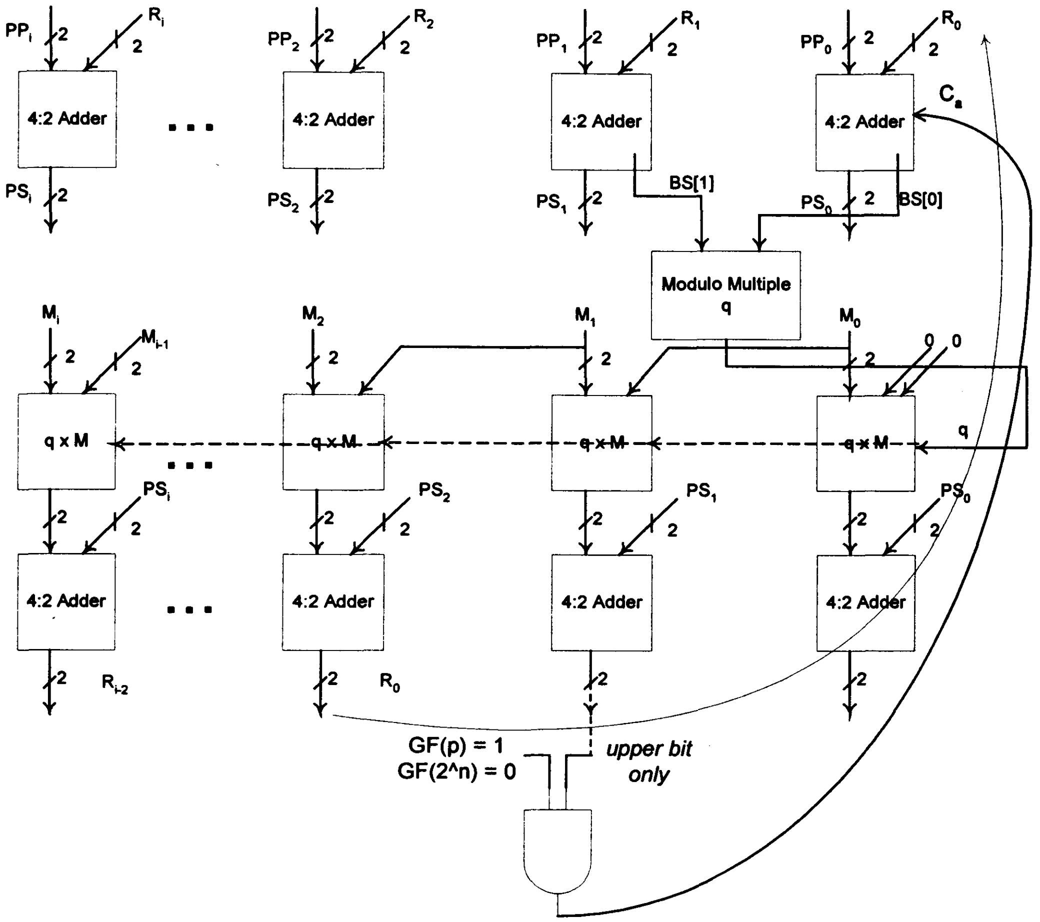


Figure 5.20 Modified architecture with carry absorption

However, if the carry is absorbed this way, then $C_a[0]$ may no longer stay as 0 which would affect the simplified design of the Montgomery modular reduction module. A test unit can be implemented to check if value 2 will appear in the addition of the last two bits of $R_0 = PS + q_iM$.

Since the input A could only be of value {0} or {1}, the last two bits of the partial product PP formed by $A*b_i$ could only be of value {00}, {01}, {10}, {11} and {21}, as shown in Table 5.7.

Table 5.7 Possible results *PP* = *A*b$_i$*

A_{i+1}, A_i	00	01	10	11
PP = A*bi	*0 = 00	*0 = 00	*0 = 00	*0 = 00
	*1 = 00	*1 = 01	*1 = 10	*1 = 11
	*2 = 00	*2 = 10	*2 = 00	*2 = 10
	*3 = 00	*3 = 11	*3 = 10	*3 = 21

Table 5.8 shows all possible result of the last two bits of *PS* = *PP* + *R*, verified by simulation. The possible results are limited to {00}, {01}, {10}, {11} and {20}.

Table 5.8 All possible PS = PP+ R for GF(p)

PS	R[1:0]									
		00	01	02	10	11	12	20	21	22
PP[1:0]	00	00	01	10	10	11	20	00	01	10
	01	01	10	11	11	20	01	01	10	11
	10	10	11	20	00	01	10	10	11	20
	11	11	20	01	01	10	11	11	20	01
	21	01	10	11	11	20	01	01	10	11

Montgomery modular reduction requires *PS* to be added to the appropriate multiple of the Modulus, so that the last two bits of the sum become 00. Table 5.9 shows all combination of *PS* + q_iM. Table 5.8 already shows all the possible *PS* from simulation. Due to the implementation of the adder, the sum of *PS* + q_iM may not stay as 2 even though the value is equivalent to 2. Instead, a carry may be carried forward to the next bit. For example, 10 + 10 = 00 with carry and 11 + 01 = 20.

Table 5.9 Combinations of $PS + q_iM$

	PS				
	00	01	10	11	20
q_iM	+00	+11	+10	+01	+00
carry	no	yes	yes	Yes	yes

Table 5.10 shows the $q*M_i$ results for all possible cases of PS (2 LSB only) for when M =01 or 11. Table 5.9 shows all the results of PS + $M*q_i$, note that some of the results stay in the form of {20} instead of turning into {00} as explained earlier. Also, c denotes a carry of 1 has been propagated to the next bit during the addition and d denotes a carry of 1 is required to be propagated to the next bit because of the state "2" being remained as a result of the addition. This concurred with the result from Table 5.9, when PS = {00} and no carry is produced.

Table 5.10 Result $M*q_i$ for different cases of PS

	PS_{i+1}, PS_i	00	01(0)	10(0)	11(0)	20
$M*q_i$	M = 01	*0 = 00	*3 = 11	*2 = 10	*1 = 11	*0 = 00
	M = 11	*0 = 00	*1 = 11	*2 = 10	*3 = 21	*0 = 00

Table 5.11 Results for R

PS_{i+1}, PS_i		00	01	10	11	20
$PS + M*q_i$	M = 01	+00 = 00	+11 = 20 (d=1)	+10 = 00 (c=1)	+01 = 20 (d=1)	+00 = 00 (c = 1)
	M = 11	+00 = 00	+11 = 20 (d=1)	+10 = 00 (c=1)	+21 = 20 (c=1) (d = 1)	+00 = 00 (c = 1)

Therefore unless the last two bit of PS = 00, a carry of 1 will be carried forward to the rest of addition result R, so we can do a carry-test, which is carried out at the same time as $q_i \times M$, as seen in Figure 5.21.

Carry only occurs for GF(p), therefore the carry test only needs to test for either PS = 1 (recoded as 11) or 2 (recoded as 10). Therefore only the upper bit of the two LSB of PS need to be checked.

Since the least significant two (4:2) units are basically unused, the third (4:2) unit effectively becomes the first (4:2) unit. The output of the carry test unit is then connected to the C_a input of the (4:2) adder. Note that in the case when $PS = \{11\}$ and $M = \{11\}$, two different carries are produced. The second carry can be absorbed by connecting it to signal C_b.

The implementation of the carry test module is shown in Figure 5.22, it is implemented as an OR gate for C_a and an additional 3-input NAND gate for C_b.

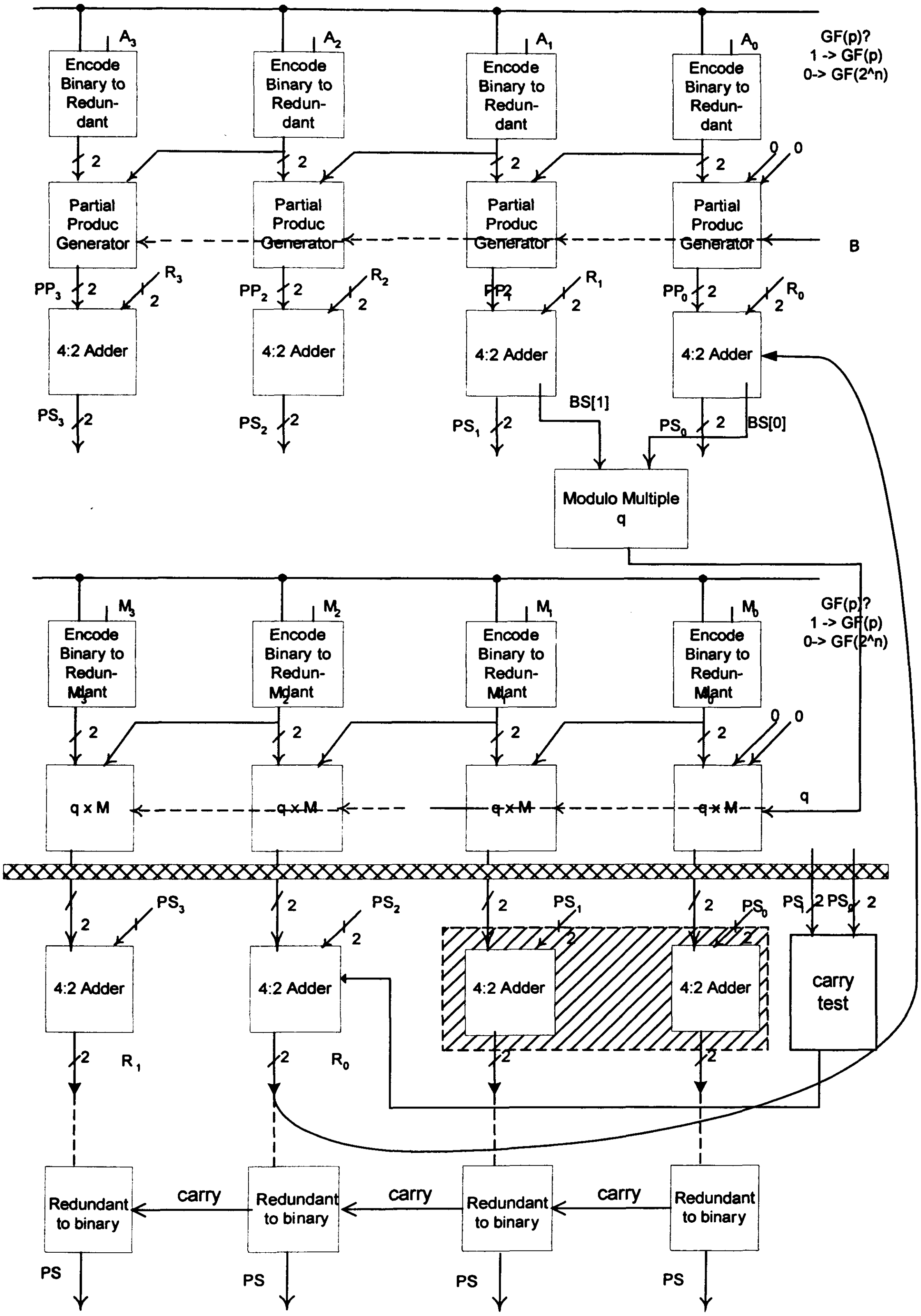


Figure 5.21 Overall architecture with carry test

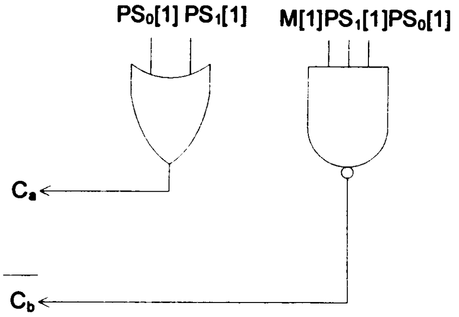


Figure 5.22 Carry Test

5.3.4 Redundant to Binary Number Conversion

The last step is to convert the final result of the multiplication from redundant representation back to binary numbers. Table 5.12 shows the conversion of the representations. 00 represent 0; both 01 and 11 represent 1 and 10 represents 2, which mean 0 with a carry. Table 5.13 shows all the details of the conversion. The binary value of each bit is not just based upon its value alone; it is also dependent on the carry bit from the next least significant redundant digit.

Table 5.12 Redundant to binary representation

		Binary representation	
		SUM	CARRY
Redundant representation	0	0	0
	1	1	0
	1*	1	0
	2	0	1

Table 5.13 Redundant to binary conversion with Carry

Bit *i*		Bit *i*-1				
				Carry from		Carry to bit
X_1	X_0	Y_1	Y_0	bit *i*-1	bit *i*	*i*+1
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	X	X	X
0	1	1	1	X	X	X
1	0	0	0	0	0	1
1	0	0	1	0	X	X
1	0	1	0	1	1	1
1	0	1	1	0	0	1
1	1	0	0	0	1	0
1	1	0	1	0	X	X
1	1	1	0	1	0	1
1	1	1	1	0	1	0

Table 5.14 Karnaugh map for binary bit conversion

		Bit *i*			
		0 (00)	1* (01)	1 (11)	2 (10)
	0 (00)	0	1	1	0
Bit *i*-1	1* (01)	0	1	X	X
	1 (11)	0	X	1	0
	2 (10)	1	X	0	1

Table 5.15 Karnaugh map for carry bit generation

		Bit *i*			
	Carry	0 (00)	1* (01)	1 (11)	2 (10)
Bit *i*-1	0 (00)	0	0	1	1
	1* (01)	0	0	X	X
	1 (11)	0	X	0	1
	2 (10)	0	X	1	1

The actual value of the previous redundant representation is not essential; the information that is required from the previous redundant bit is whether or not it will provide a carry. That is when the radix-4 number is {10} which represents 2, the carry bit is 1. Therefore, Table 5.16 will be more appropriate. Table 5.17 and Figure 5.18 show the relevant Karnaugh map.

Table 5.16 Redundant to binary conversion by checking carry from bit *i*-1

*x*_1	*x*_0	Carry from bit *i*-1	Bit *i*	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	x	X
1	0	0	0	1
1	0	1	1	1
1	1	0	1	0
1	1	1	0	1

Table 5.17 Karnaugh map for binary conversion by checking carry from the bit *i*-1

		Bit *i*			
	Binary bit	0 (00)	1* (01)	1 (11)	2 (10)
Carry from bit *i*-1	0	0	1	1	0
	1	1	x	0	1

Table 5.18 Karnaugh map for carry bit generation by checking carry from the bit *i*-1

		Bit *i*			
	Carry	0 (00)	1* (01)	1 (11)	2 (10)
Carry from bit *i*-1	0	0	0	0	1
	1	0	x	1	1

Because of the way the numbers are coded, the lower bit of '0' and '2' are both '0' and the lower bit for '1'and '1*' are both one, the upper bit effectively provides the field information, apart from the state '0', which shares the same characteristic in both state. Therefore, for binary conversion, only the lower bit needs to be considered, along with the carry bit from the previous redundant representation. For the generation of the carry bit to the next bit, only three cases will produce a carry:

- state '2' with no carry from bit *i*-1
- state '2' with carry from the bit *i*-1

- state '1' with carry from the bit *i*-1

Therefore, each binary conversion module consists of the following circuit as shown in Figure 5.23. Figure 5.24 shows the overall architecture of the multiplier.

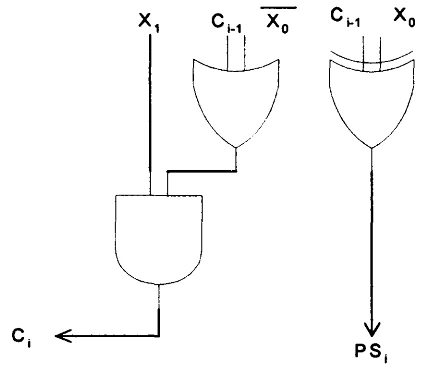


Figure 5.23 circuit for binary conversion

This chapter has presented a radix-4 unified field digit-serial Montgomery multiplier, which includes:

- A novel (4:2) adder for unified GF(*p*) and GF(2^n) Galois Field Multiplication.
- A partial product generator to work with the proposed (4:2) adder
- A Montgomery reduction module

At the beginning of this chapter, common modulo reduction methods were discussed and it was shown that Montgomery's algorithm is considered to be one of the most efficient algorithms around and is particularly suitable for the scope of this design. The implementation of individual modules was then discussed. The redundant adder described in Chapter 4 was used to reduce the long carry chain. The main difference in implementation between the proposed idea and other previous research is that the information regarding the Galois Field under which the addition is to be performed is embedded into the digit coding, so that there is no need for a globally-broadcast control signal.

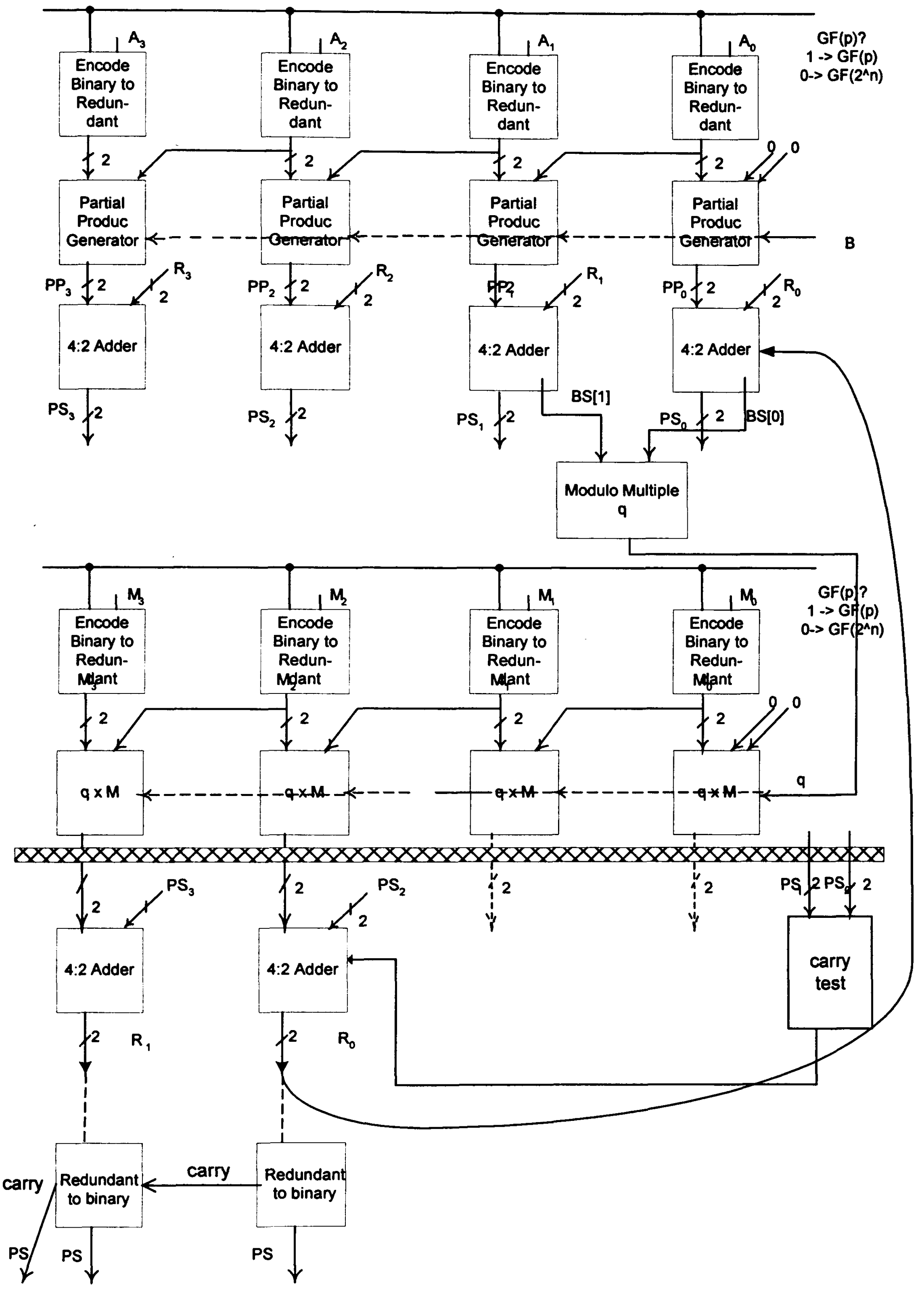


Figure 5.24 Final overall architecture

In this Chapter, the proposed radix-4 unified field digit-serial Montgomery multiplier was shown. This multiplier is:

- Scalable – the proposed design embeds the field information into the digit coding such that it does not allow the FSEL delay dominates the pipeline stage. (4:2) redundant adder is used to remove carry propagation.

- Fast - This design is particularly suitable for long word length. It does not have the problem that FSEL buffer delay starts to dominate the critical path and affect the maximum clock rate achievable. This design can be pipelined to enhance the performance.

- Impartial - Both GF(p) and GF(2^n) operations can be carried out equally easily.

In the next chapter, the area and the delay in terms of Logical Effort will be assessed. Methods to improve the overall performance of the multiplier will be introduced, which include hardware optimisation and the use of quotient pipelining. Other improvements and further research possibilities will be discussed and finally, the conclusions from this research will be drawn.

6 Comparisons, Improvements and Conclusions

Chapter 5 presented a radix-4 unified field digit-serial Montgomery multiplier, which included:

- A novel (4:2) adder for unified GF(p) and GF(2^n) Galois Field Multiplication.
- A new partial product generator to work with the proposed (4:2) adder
- A Montgomery reduction module adapted for redundant digits

In this chapter, the design of the unified field multiplier will be assessed and compared with the previously proposed design. Chapter 4 showed that the proposed redundant adder design is very much more scalable compared with that of Savaş *et.al.* [121]. This is due to the fact that in the proposed design, instead of broadcasting the field information throughout the whole circuit, it is encoded within the number itself. Scalability of a cryptographic processor is important because long word-length multiplication is often required. Moreover, from the security point of view, the larger the cryptographic operation processor, the less susceptible it is to differential power attacks [1], therefore it is beneficial to make the design as large as possible without compromising on the performance. The scalability of the partial product generator will be assessed in the first part of this chapter. The use of quotient pipelining will be introduced as a means of improving the performance of the multiplier. Conclusions will be drawn in the third part of this chapter; improvement methods that could also be applied to this radix-4 hardware multiplier will also be discussed.

6.1 Overall unified field multiplier assessments

The multiplier comprises four different modules: (1) Binary to Redundant number encoder (AND gate); (2) Partial Product Generator; (3) (4:2) adder for partial products summation and modulo reduction; (4) Modulus Multiplier Digit Selection. These designs have been presented in the previous Chapters. The speed of the (4:2) adder has been assessed using Logical Effort so the other modules will also be evaluated using this method. However, the Binary to Redundant number encoder is not on the critical path of the multiplier so there is no need to calculate its delay.

6.1.1 Area and Speed of Partial Product Generator

The gate count of the partial product generator for *w*-bit operands including the binary to redundant number encoder is as follows:

- 2*w* AND
- 2*w* 4-input MUX
- 1*w* OR
- 1*w* XOR

The partial product generator processes the multiplicand in radix-4 format, i.e. two bits are scanned per iteration. This halves the number of iterations required; however, the area of the circuit is compromised. A traditional 4-input multiplexer implementation is shown in Figure 6.1, but improvement can be made by implementing each 4-input multiplexer as 4-way tri-state inverter [131], as shown in Figure 6.2 and Figure 6.3.

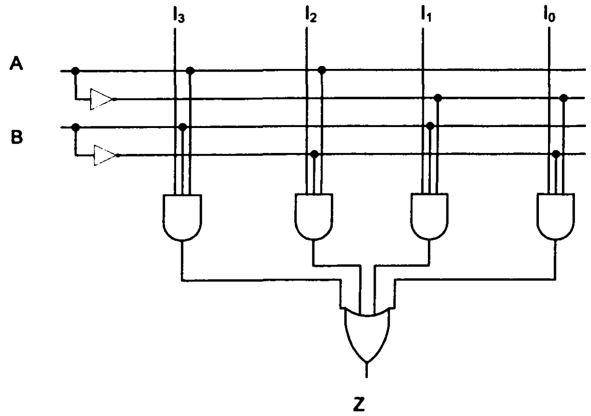


Figure 6.1 Traditional MUX implementation

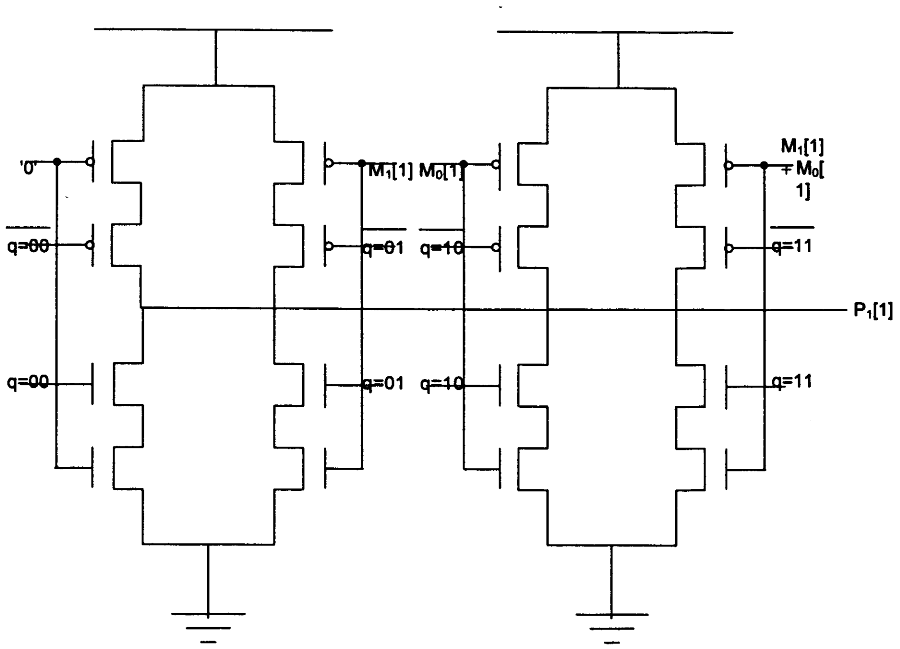


Figure 6.2 MUX implementation for $P_1[1]$ as a 4-inputtri-state inverter

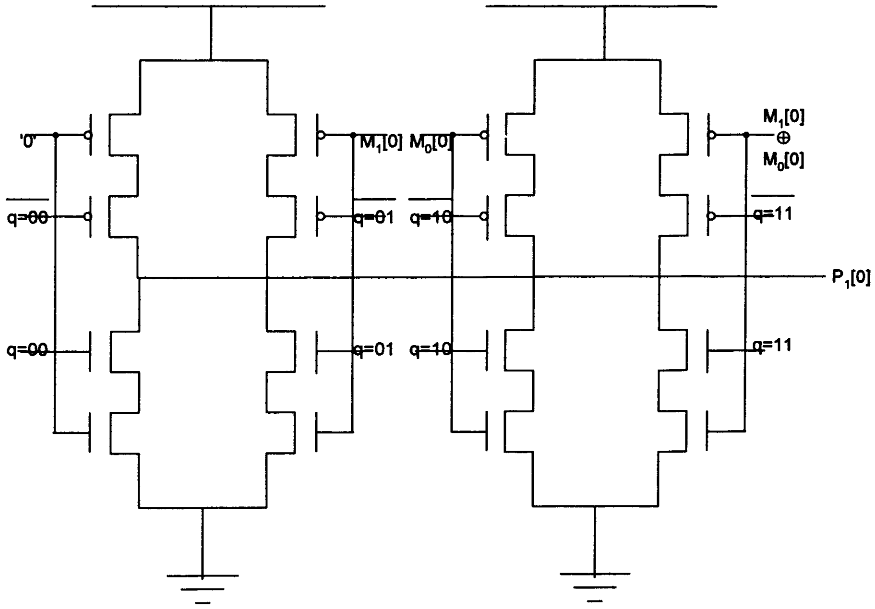


Figure 6.3 MUX implementation for $P_1[0]$ as a 4-input tri-state inverter

Figure 6.4 shows the overall multiplexer input connection of the modulus multiplication q_iM. This also shows that instead of using unencoded multiplier and multiplicand operands as inputs, and then decoding them using the 3-input NAND gate implementation of multiplexers, it is more efficient to use the encoded multiplier and multiplicand as inputs along with the tri-state inverter implementation of the multiplexer. The same layout can be applied to the partial product generation $A \times B$. The timing and scalability assessment of the partial product generator using Logical Effort is discussed in Section 6.1.2.

The 4-way inverting multiplexer consists of four transistor-select arms, each to be selected by respective selection signal:

- s*1 – multiplier A x B = 00, i.e. q_i[1] NOR q_i[0]
- s*2 – multiplier A x B = 01, i.e. (NOT q_i[0]) NOR q_i[1]
- s*3 – multiplier A x B = 10, i.e. (NOT q_i[1]) NOR q_i[0]
- s*4 – multiplier A x B = 11, i.e. (NOT q_i[1]) NOR (NOT q_i[0])

Each of the respective data input is connected to the output such that when an input is selected, the relevant bundle will then be driven to state TRUE.

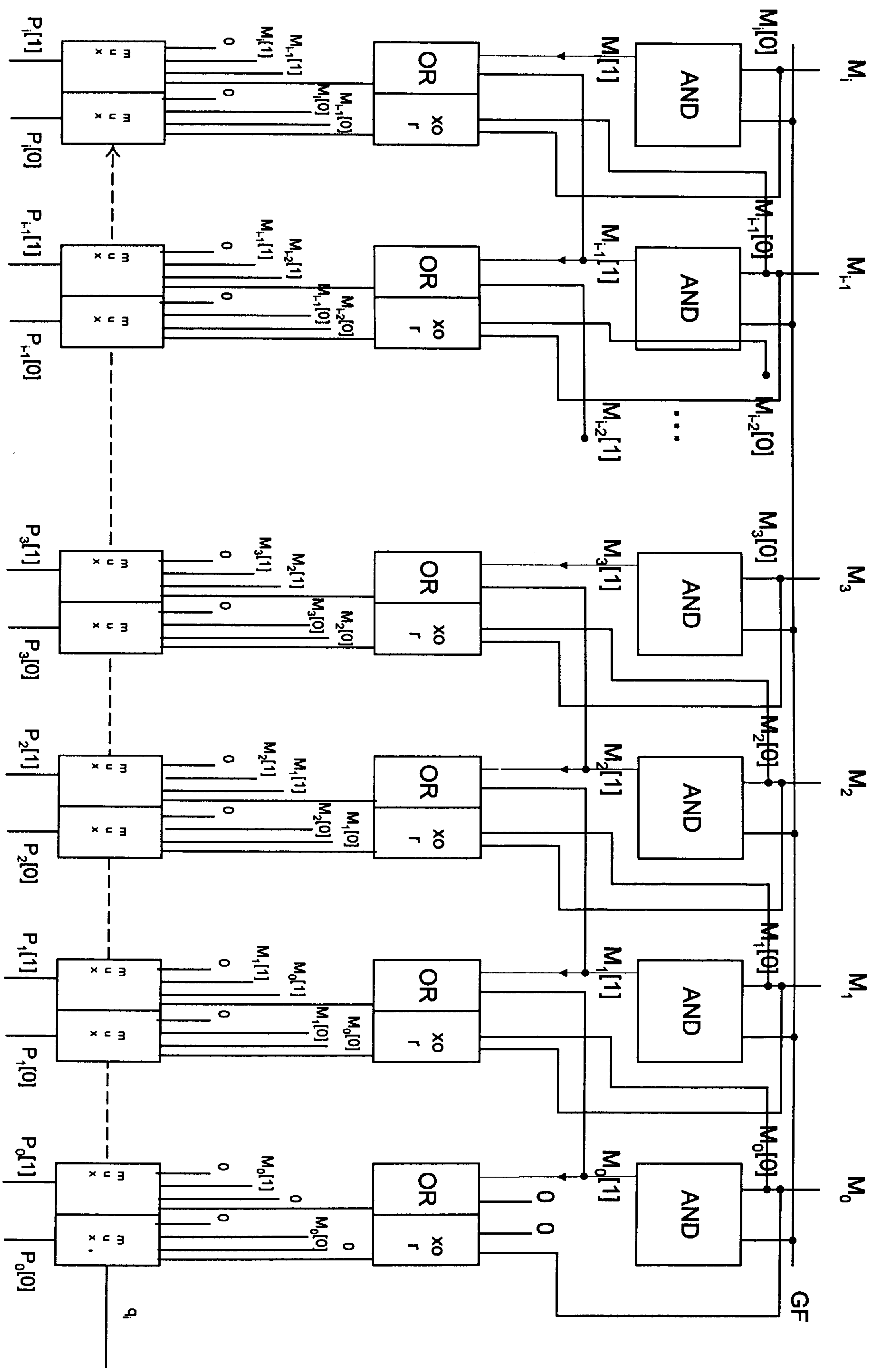


Figure 6.4 Multiplexer input connection for q_iM

The gate count of the modulo multiple *q* selection module (see Figure 5.18) is as follows:

- 3 XOR
- 1 2-input MUX

This small module forms irregularity on the critical path, therefore effort is made to ensure data is derived as efficiently as possible; details are discussed in Section 5.3.2. Since there is an individual module that does not need to be replicated, the size of the module becomes insignificant compared with the rest of the circuit. This module operates in radix-4 mode such that the effect of the bottle-neck on the critical path is minimised.

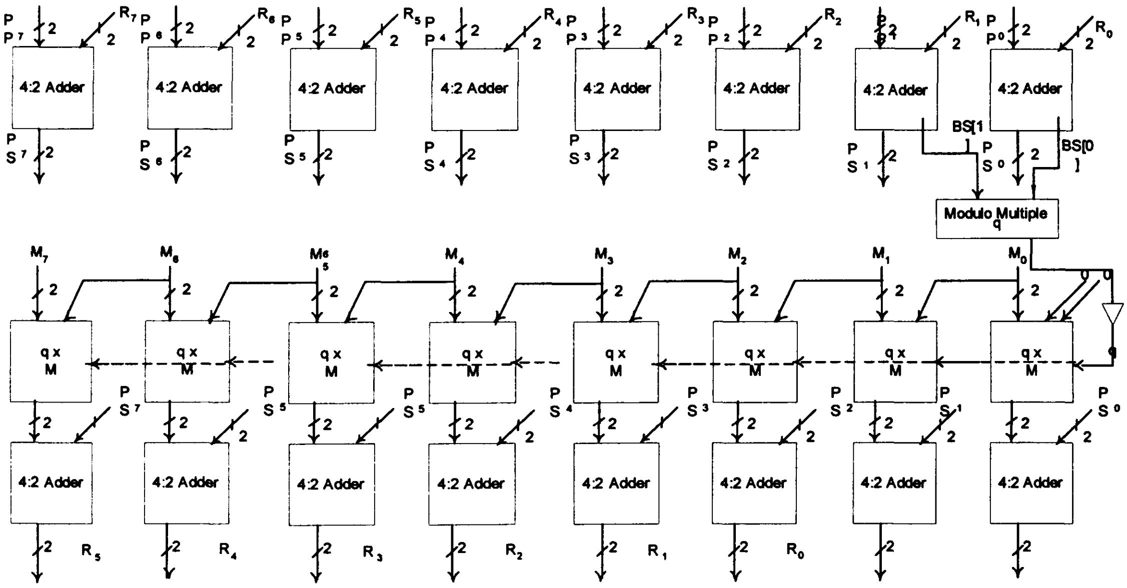


Figure 6.5 8-bit Multiplier simulation diagram

The 8-bit multiplier shown in Figure 6.5 was simulated using NC-Verilog and synthesised using Synopsys, which showed that the critical path from the input to the first row of adders to the output of the second row of adders was 3.77 ns including 0.31 ns register delay in 0.18μm VLSI technology (See Appendix 3 for timing report.). Note that the multiplexer implementation used for the purpose of this synthesis was in the form of traditional "AND – OR" gate structure as shown in Figure 6.1. Section 6.1.2 will assess the implementation of multiplexer using tri-state inverters as shown in Figure 6.2. Section 6.1.2 will show how this implementation can provide an improvement on the multiplexer delay.

6.1.2 Overall unified field multiplier assessment - Scalability

In section 4.3, the scalability of the unified field adder was discussed, it was concluded that (4:2) unified field adder provide the ability to increase the word length of the adder without causing extra delay due to high fanout, which means the proposed adder is truly scalable. This ability is granted by absorbing the field information into the coding such that the field select signal does not need to be broadcast through the adder.

In terms of the overall multiplier design, there are four occasions where a signal is to be propagated along a chain of modules, which could affect the scalability of the multiplier due to high fanout, they are:

1. Field information to multiplier A for partial product generation
2. Multiplicand digit b_i propagation for partial product generation
3. Field information to modulus M for Modulus multiplication
4. q_i propagation for modulus multiplication

Situations 1 and 3 do not impose any severe threats to the overall propagation delay, because both multiplier A and modulus M stay constant while the multiplicand digit b_i and the modulo multiple q_i change every iteration. The possible delay caused by long propagation of multiplicand digit b_i does not affect the critical path because the output is well latched till it is required for the next computation. However, the delay is affected by the modulo multiple q_i which propagates along two times the word-length worth of multiplexers, as shown in Figure 6.6. Later in this chapter we will show how this situation can be rectified by applying appropriate pipelining.

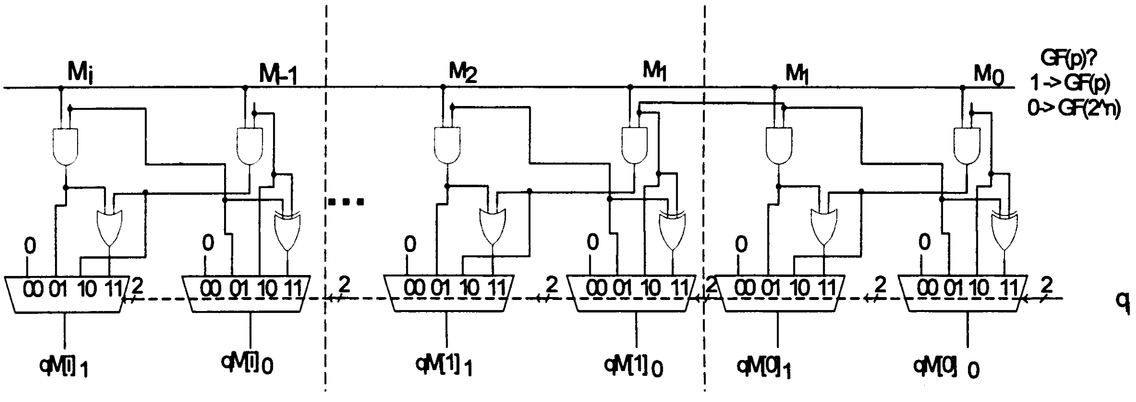


Figure 6.6 PPG unit quotient q_i propagation

Figure 6.7 shows the implementation of a multiplexer required for partial product generation. The input load is connected to an inverter and two 3-input NAND gates. Therefore, the branching effort (b) for q_0 and q_1 is (1 + 5/3 + 5/3 + 3×2/3) = 19/3 per multiplexer. Each partial product generation module has 2 multiplexers, therefore the total branching effort b of the partial product generation module is (19/3 × 2) = 38/3 per bit. The calculation of the Logical Effort for the partial product generator is shown in Table 6.1.

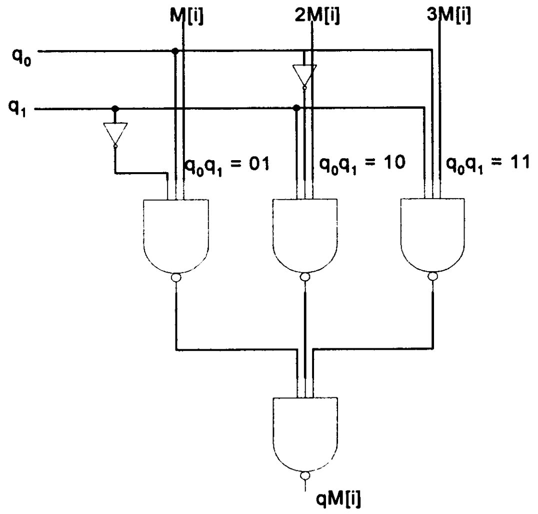


Figure 6.7 MUX implementation

Table 6.1 4 input MUX logical effort

	Logical effort g	Branching effort b	Electrical effort h	Parasitic effort p	Path effort $gb(h)$
Input	1	38/3 w	1	-	38/3 w
NOT	1	(5/3 + 2/3) / (5/3)	1	1	7/5
3 - NAND	5/3	(5/3 + 2/3) / (5/3)	1	3	5/3
3 - NAND	5/3	(2 + 4 + 4/3) / 4 (AOI + XOR)	1	3	55/18

Notice that the output of the final 3-input NAND gate is connected to the (4:2) redundant adder, which means it is connected to an AOI and an XOR gate for the worst case. The total path effort of the critical path for w word length is: $F = GBH = (w \times 38/3) \times 7/5 \times 5/3 \times 55/18 = 90.3\ w$.

The number of stages N needed including buffers can be calculated as follows:

If $w = 1$ bit:

$N \quad = \text{Rnd}\ (\log_4 F)$

$= \text{Rnd}\ (\log_4 90.3) = \text{Rnd}(\ \ln (90.3)/ \ln 4\ ) = \text{Rnd}\ 3.25$

$\therefore N = 3$

This means that 3 stages should be included in the path and therefore no buffers need to be added. The stage load/drive α is calculated as follows:

$\alpha \quad = F^{1/N} = 90.3^{\ 1/3} = 4.49$

Finally, the delay along the critical path D is defined as $D_{critical} = (N \times \alpha + P)/5$ in FO4 unit.

$D \quad = (3\times4.49 + 7)/5 = 4.09$ FO4

The delay of the 4-input multiplexer for different wordlength implemented using NAND gates is summarised in Table 6.4.

Table 6.2 Logical effort delay for the 4-input multiplexer (traditional implementation) of different wordlength

Wordlength (bit)	$\alpha \quad = F^{1/N}$	$D \quad = (N\alpha + P)/5$ (FO4)
4	$(90.3 \times 4)^{1/4} = 4.36$	$(4 \times 4.36 + 7 + 1)/5 = 5.09$
16	$(90.3 \times 4 \times 4)^{1/5} = 4.29$	$(5 \times 4.29 + 7 + 2)/5 = 6.09$
64	$(90.3 \times 4 \times 4 \times 4)^{1/6} = 4.24$	$(6 \times 4.24 + 7 + 3)/5 = 7.08$
256	$(90.3 \times 4 \times 4 \times 4\times 4)^{1/7} = 4.20$	$(7 \times 4.20 + 7 + 4)/5 = 8.08$

Figure 6.8 is a graph showing the relationship between wordlength of the multiplexer chain vs. the delay from q_i to q_iM in FO4 for both traditional multiplexer implementation and tri-state inverter implementation. There is a logarithmic relationship between the wordlength and the delay, whereby increasing w by a factor of 4 adds only 1 FO4 to the delay of buffering q_i across the w-bit multiplexer.

Recall in section 4.3.2 shows that design by Savaş *et. al.* has restriction on the word length of adder before the delay caused by the field signal broadcast will dominate the critical path. The proposed design is an improvement in comparison. However, further improvement can be made by implementing the 4-input multiplexer as four tri-state inverters as mentioned in Section 6.1.1.

The logical effort on each input for this 4-to-1 multiplexer is (4+2)/3 = 2, plus 2/3 for wire gives the total logical effort of 8/3 per bit. This is done by bundling the complementary selection input "ab = xx" and "$\overline{ab}$ = xx" together. The parasitic delay p is equal to 8. Whereas for the case using the 3-input NAND gate implementation, the logical effort is equalled to 38/3 per bit. This can be summarised as shown in Table 6.3.

Table 6.3 Logical effort for the 4-input multiplexer (4-input tri-state inverter implementation)

	Logical effort g	Branching effort b	Electrical effort h	Parasitic delay p	Path effort $gb(h)$
Input	1	(8/3 × 2 × w) / 2 (2 muxes per bit)	1	-	8/3 w
MUX-4	2	(2 + 4 + 4/3) / 4 (AOI + XOR)	1	8	22/6

Path Effort of the critical path $F = gb(h)$ (critical path) = (w × 8/3) × 22/6 = 9.78 w, compared with 90.3 w as in the previous case, there is a nine-fold improvement.

The number of stage N needed including buffers can be calculated as follows:

If w = 1 bit:

N = Rnd ($\log_4 F$)

= $\log_4 9.78 = \ln(9.78)/\ln 4 = 1.64$

Rnd 1.64 = 2

$\therefore N = 2$

Therefore, there is no need for extra buffer, only 2 stages are needed. The stage load/drive α and the delay D is calculated as follows:

$\alpha = F^{1/N} = 9.78^{1/2} = 3.13$

$D = (N \times \alpha + P)/5 = (2\times3.13 + 8)/5 = 2.85$ FO4

The delay of the 4-input multiplexer for different wordlength is summarised in Table 6.4.

Table 6.4 Logical effort delay for the 4-input multiplexer (tri-state inverter implementation) of different wordlength

Wordlength (bit)	$\alpha \quad = F^{1/N}$	$D \quad = (N\alpha + P)/5$ (FO4)
4	$(9.78 \times 4)^{1/3} = 3.39$	$(3 \times 3.39 + 8 + 1)/5 = 3.84$
16	$(9.78 \times 4 \times 4)^{1/4} = 3.54$	$(4 \times 3.54 + 8 + 2)/5 = 4.83$
64	$(9.78 \times 4 \times 4 \times 4)^{1/5} = 3.62$	$(5 \times 3.62 + 8 + 3)/5 = 5.82$
256	$(9.78 \times 4 \times 4 \times 4 \times 4)^{1/6} = 3.68$	$(6 \times 3.68 + 8 + 4)/5 = 6.82$

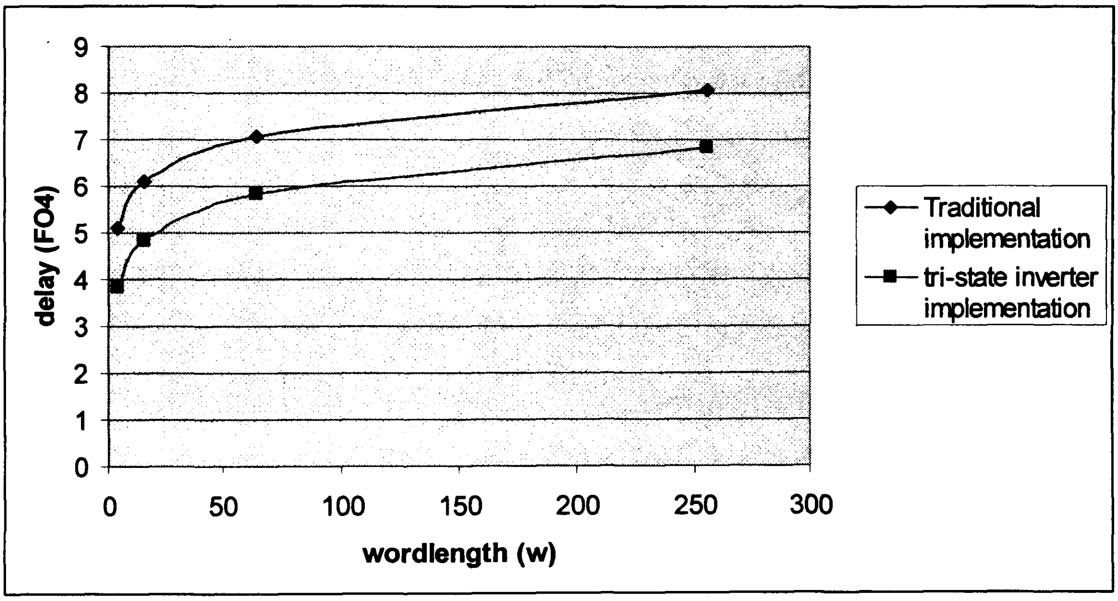


Figure 6.8 Wordlength *w* vs. Delay (FO4) for traditional and tri-state inverter multiplexer implementation

Figure 6.8 shows the relationship between the wordlength and the delay of the multiplexer. It shows the improvement of delay by using the tri-state multiplexer compared with the traditional implementation, in fact the tri-state multiplexer improves the speed by 1.2 FO4 for all word-lengths.

6.1.3 Area and Speed of Modulus Multiplier Digit Selection

Finally, the delay of the modulus multiplier digit selection has to be found. Section 6.1.1 described the implementation of the tri-state inverter including the four selection signals for the 4-input tri-state inverter. Table 6.5 shows the delay of the modulus multiplier digit selection, considering the input $B[1]$ is connected to an XOR and a 2-input multiplexer, allowing 2×2/3 for wires, the total branching effort is 22/12. This is then connected to a NOR gate for encoding of tri-state inverter selection.

Table 6.5 Modulus Multiplication Digit

	Logical effort *g*	**Branching effort *b***	**Electrical effort *h***	**Parasitic effort *p***	**Path effort *gb(h)***
Input *B*[1]	1	(4 + 2 + 4/3) / 4	1	-	22/12
XOR	4	(2 + 2/3) / 2	1	4	16/3
MUX	2	(5/3 + 5/3 + 1 + 6/3) / (5/3)	1	4	38/5
NOR	5/3	2 (*q* input to tri-state MUX)	1	2	10/3

Path Effort of the critical path $F = GBH$ (critical path) = 247.7.

The number of stages N needed including buffers can be calculated as follows:

N = Rnd ($\log_4 F$)

$= \log_4 247.7 = \ln(247.7)/\ln 4 = 3.98$

Rnd 3.98 = 4

$\therefore N = 4$

However, there are 5 stages in the circuit because XORs have two stages.

$$\alpha \quad = F^{1/N} = 247.7^{1/5} = 3.01$$

$$D \quad = (N \times \alpha + P)/5 = (5 \times 3.01 + 10)/5 = 5.01 \text{ FO4}$$

The combined delay of the partial product generator and multiplier digit selector for w bits is 2.85 + $\log_4 w$ + 5.01 = $\log_4 w$ + 7.86. This delay is greater than the (4:2) adder and so methods are needed to minimise the impact of it.

6.2 Quotient pipelining

Another method to improve the performance of the multiplier is by quotient pipelining the architecture. Quotient pipeline was introduced by Shand [153] and further developed by Orup [154]. They presented a variant of the Montgomery's algorithm such that the determination of the quotient becomes trivial and the cycle time becomes independent of the choice of radix. The idea is to delay the use of quotient digit q_{i-d}, determined from information available in iteration i-d by d iterations. This method is also used in designs such as [155] and [156].

The basic algorithm and the structure of the Montgomery multiplier presented so far are based on the algorithm shown in Figure 6.9 and architecture shown inFigure 6.10. The dependency of the modulus multiple selection unit to the last two non-redundant bit of the addition of (R + b_iA) becomes the bottleneck to the design of the pipelined structure.

```
Algorithm 1: MonPro1 ($A$, $B$, $M$) (radix-2)

MonPro1 ($A$, $B$, $M$)
{
        $R_{-1}$ := 0;
        for $i = 0$ to $n$-1 do
                $q_i := (R_{i-1} + b_iA)$ Mod 2;
                $R_i := (R_{i-1} + q_iM + b_iA)/2$;
        end for
return $R_{n-1}$;
}
```

Figure 6.9 Algorithm 1 non-pipelined

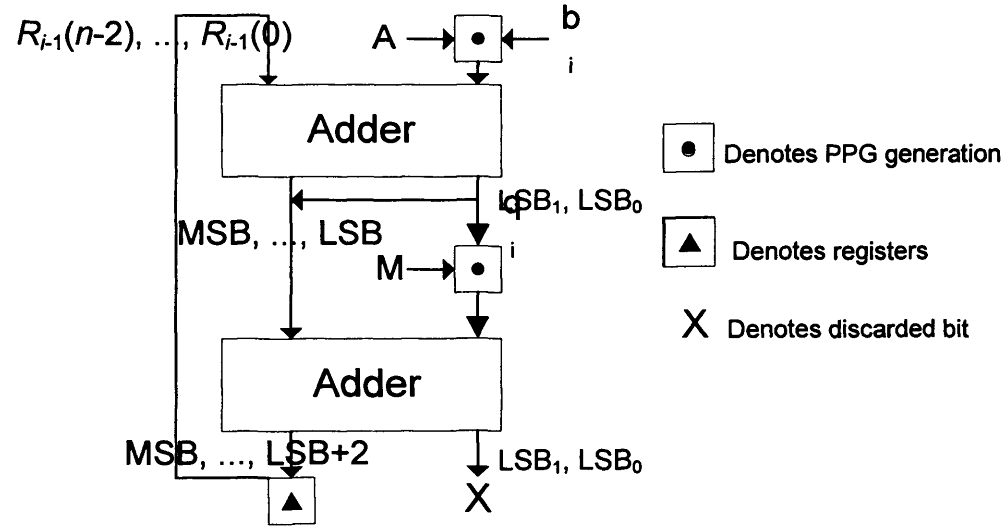


Figure 6.10 Architecture 1 non-pipelined

The quotient selection process is dependent on $BS[1]$ and $BS[0]$ from the first row of (4:2) adders, these are formed from the addition of the partial product $A \times B_i$ and the partial result from the previous iteration. In order to pipeline this structure efficiently, the quotient pipelining method forces the last bit (in a radix-2 design), or in this case, the last two bits of A to 0, since this design is a radix 4 operation. By doing so, the two binary bits that are inspected for determining the quotient bit will only be dependent on the partial result from the previous iteration, thus removing this logic from the critical path. The algorithm for radix-2 operation and the architecture for the proposed radix-4 multiplier are shown in Figure 6.11 and Figure 6.12 respectively.

```
Algorithm 2: MonPro2 ($A, B, M$)

MonPro2 ($A, B, M$)
{
        $S_{-1} := 0$;
        $A := 2 \times A$;
        for $i = 0$ to $n$ do
                $q_i := (S_{i-1})$ Mod 2;
                $S_i := (S_{i-1} + q_i M + b_i A) / 2$;
        end for
return $S_n$;
}
```

Figure 6.11 Algorithm 2 with quotient pipelining for radix-2 multiplication

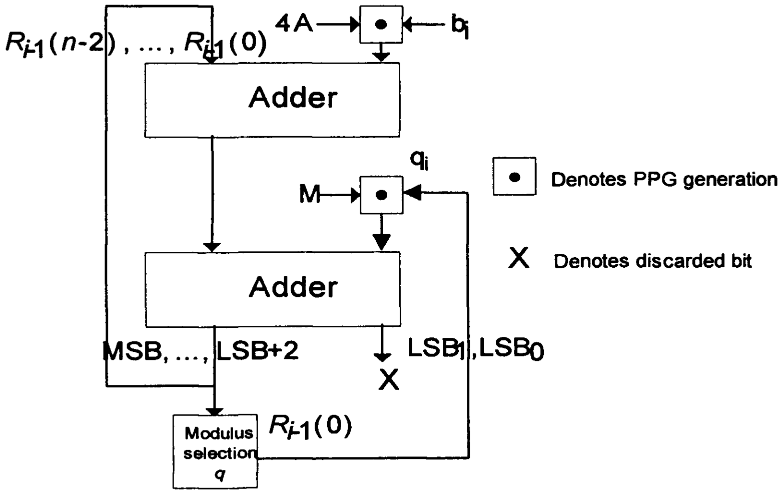


Figure 6.12 Proposed architecture with quotient pipelining (for radix-4 multiplication)

Figure 6.13 shows how quotient pipelining based on Algorithm 2 shown in Figure 6.11 can be incorporated into the proposed unified field multiplier.

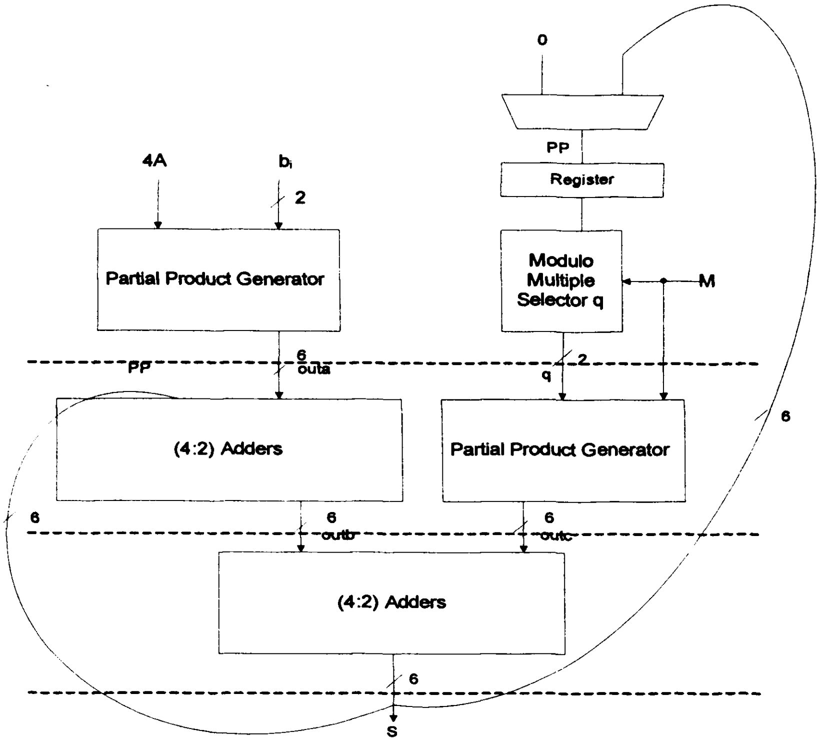

0
PP
Register
4A
bi
2
Partial Product Generator
Modulo Multiple Selector q
M
pp
6
outa
2
q
(4:2) Adders
Partial Product Generator
6
6
6
outb
6
outc
(4:2) Adders
6
S

Figure 6.13 Quotient Pipelined multiplier architecture

This procedure requires one extra cycle of pre-processing and one extra cycle of post-processing to remove the effects of the extra factor of 4. The quotient pipelined version described above was simulated with $w = 8$ using NC-Verilog and synthesised using Synopsys, which showed that the critical path which now only goes through the (4:2) adders was 3.20 ns (worst case + register set up) using 0.18μm VLSI technology. This is 0.57 ns faster than the original design because the q_iM logic has been removed from the critical path. At larger values of w the speed-up is even more apparent.

Figure 6.13 shows that the q_i selector and q_iM partial product generator operate in parallel with the first (4:2) adder. Since the delay of the (4:2) dual field redundant adder is 5.80 FO4 and the selector and generator have a combined delay of 7.86 + $\log_4 w$ FO4, quotient pipelining does not completely "hide" the delay of the buffering.

Most delay is due to partial product generator (2.85 + $\log_4 w$ FO4) instead of q_i selector (5 FO4) for $w > 16$. Taking the critical path as the PPG and the two (4:2) adders, the delay in FO4 calculated using logical effort would be: $(2.85 + \log_4 w + 2 \times 5.8) = 15.95$ FO4 for $w = 8$.

Critical path does not go through q_i every cycle because there is only a need to work out q_i once for each multi-precision word-serial multiplication. Since critical path doesn't go through q_i–selection unit every cycle, as a new q_i, it is only needed to be worked out once for each multi-precision word-serial multiplication. Therefore given the size of the multiplier as L and the wordlength of A as w, the q_i–selection unit is only used once in every w/L iterations. Therefore the critical path lies on the two dual field (4:2) redundant adders. The domination of the (4:2) adder delay will be shown clearly in the $M \times M$ example.

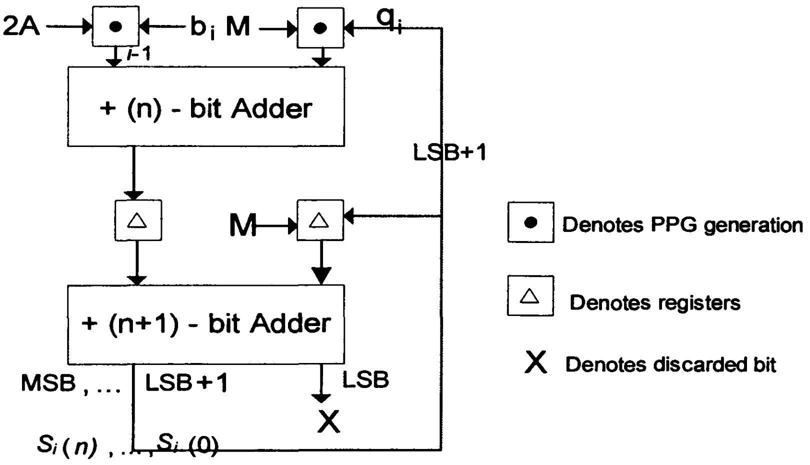


Figure 6.14 Daly's modified architecture

Daly and Marnane [156] suggested the idea of rearranging the order of the additions, as shown in Figure 6.14, such that $2A \times B$ can be parallelised with q_iM. The number of cycles required will be the same as that of the previous version where n+1 clock cycles are required. Daly and Marnane [156] stated that there is a significant increase in operation speed compared with the previous architecture because the summation of the partial product $2Ab_i$ (radix-2 case) and the modulus multiple q_iM can be performed as soon as the LSB of the other addition are complete. However, due to the fact that in the proposed radix-4 design, the modulus multiple selection process is more complex, the partial product generation of q_iM can only be carried out after the modulus multiple selection is done. In the case of radix-2, there is only a need to choose multiple of 0 or multiple of 1, whereas in the proposed radix-4 case, the modulus multiple selection unit has to decide between ×0, ×1, ×2 and ×3, see Figure 6.15. Therefore, this reconfigured version of pipeline, does not affect the proposed design dramatically as in the case of Daly and Marnane [156]. The main benefit is that it provides a more symmetrical architecture. In fact, the original pipelined architecture as shown in Figure 6.13 has only got 3 pipeline stages, whereas the proposed multiplier using Daly's quotient pipeline structure consists of 4 pipeline stage, therefore, this new structure is not suitable for the proposed design. In Figure 6.13, the effect of buffering of q_iM partial product generation is covered up by the delay of the (4:2) addition; this is not the case in Figure 6.15. Therefore if Figure 6.15 is used, the width of the partial product generation will seriously affect not just the scalability but also the delay. Another disadvantage of this pipeline architecture is that the partial product generation modules can not be reused since they are operated on simultaneously.

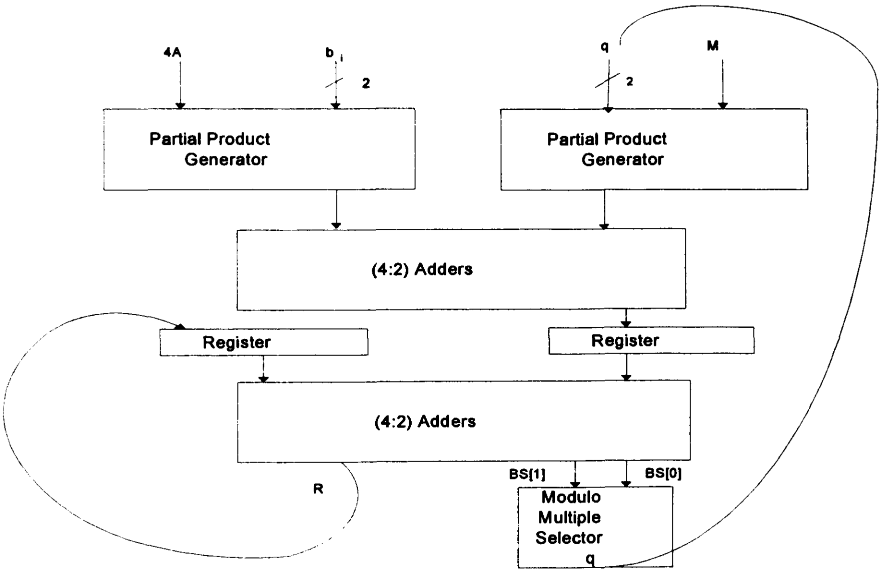


Figure 6.15 Proposed architecture using Daly's quotient pipelined structure

6.3 M-bit × M-bit multiplication

For 160-bit × 160-bit multiplication, if the size of the multiplier is of 32-bit then the operations of the multiplier is discussed as follows:

The partial product generation $A*B$ will be done in 160/32 * 160/2 cycles = 400 cycles. 32-bits of multiplier A is multiplied by 2 bits of b_i at each cycle and each b_i has to multiply 5 sets of A from bit {0...31}, {32...63}, {64...95}, {96...127}and {128...159}. 32 partial product generator modules are needed. Note that because of the special encoding system used for the unified field design, each binary bit of multiplier A is represented by two wires, therefore the register that holds $A_{\{n...n+31\}}*b_i$ is 32*2+2 = 66 bits in size.

Once the first set of partial product $PP_{\{32(n-1)... 32n-1\}}$ is generated, it can then be added to the result of the previous iteration $R_{i\{32(n-1)... 32n-1\}}$, in the meantime the next set of partial product $PP_{\{32n... 32(n+1) -1\}}$can be generated. 32 (4:2) dual field redundant adder modules are needed to perform the addition and configured as shown below. Note the *MSB* (4:2) dual field redundant adder module is to take care of the possible overflow.

In order to perform the Montgomery modulo reduction, the 2 LSB of the addition is connected to the modulus multiplier digit selection unit. Since this process is only applied to the two LSBs, therefore it can be carried out as soon as the numbers are available and can be processed in parallel with the rest of the additions.

Once q_i is computed, $q_iM_{\{32(n-1)... 32n-1\}}$ can be computed. Once again, it is generated in 5 groups of 32 bits: bit {0...31}, {32...63}, {64...95}, {96...127} and {128...159} using 32+1 = 33 sets of partial product generator modules. The extra set of *PPG* module is to take care of the possible overflow. For example, if the 2 *MSB* = {11} and b_i = 3, an over-flow will occur. Thus the results will be stored in a register that is 66 bits in size. As shown in Figure 6.16, the q_iM should only be computed only once the computation of $A*bi$ is completed. This is to avoid the need of having to have two sets of partial product generation modules.

The carry test can be carried out as soon as $PS_1[1]$ and $PS_0[1]$ are available, since the test is only an AND and a 3-input NAND gate and it is not on the critical path, it is not important when it is carried out, as long as it is done before the addition of $PS + q_iM$.

The process of the addition of $PS + q_iM$ will commence as soon as generation of PS is completed. This is to avoid the need to have two sets of (4:2) redundant adder modules; additionally one needs to make sure all the values required for this calculation are available before the computation of the addition. Figure 6.17 shows the timing diagram of the overall $M \times M$ operation, where both the adder and the partial product generator modules are reused.

Figure 6.17 shows that the addition takes up 5.8 FO4, whereas partial product generation using 4-input tri-state inverter takes only 2.85 FO4. Hence, the partial product generator finishes all the calculations for the i^{th} iteration before the (4:2) adder even completes the first half of addition. Partial product generation for the next iteration i+1 $(A*b_{i+1})$ and i+2 $(A*b_{i+2})$ can be performed before the adder is ready to compute addition for the next iteration.

Figure 6.17 also shows that the cycle delay is dependent on the delay of the two (4:2) redundant addition process. The time taken to perform the first cycle Montgomery Multiplication is $(5.8 \times 10 + 2.85 + \log_4 32)$ FO4 = 63.35 FO4. The total delay in FO4 for 160 bit × 160 bit Montgomery Multiplication is $[(5.8 \times 10) \times 160/2 + 2.85 + \log_4 32] = 4645.35$ FO4.

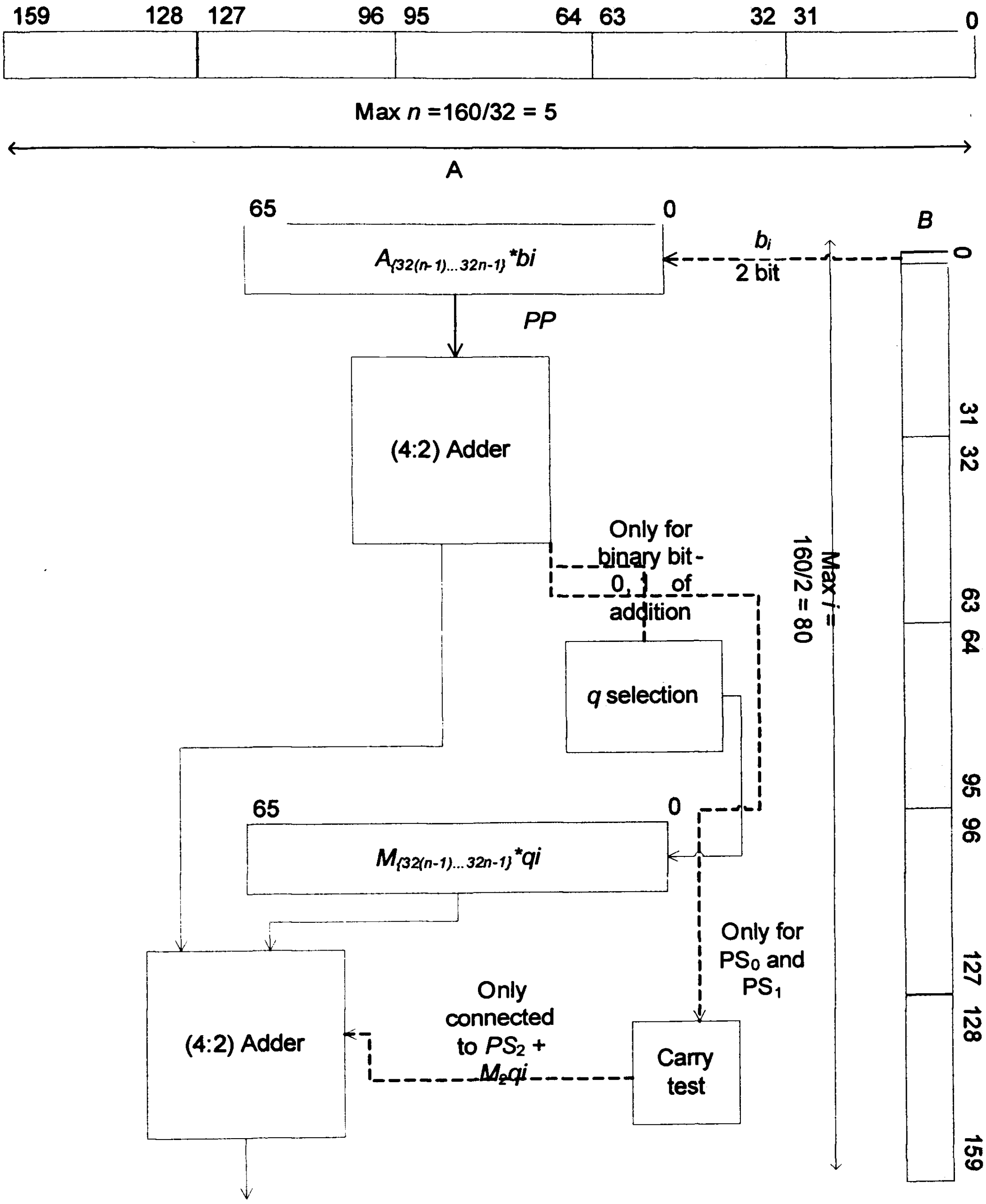

159
128
127
96
95
64
63
32
31
0
Max n =160/32 = 5
A
65
0
$A_{\{32(n-1)...32n-1\}}$*bi
b_i
2 bit
B
PP
(4:2) Adder
Only for binary bit - 0, 1 of addition
Max i = 160/2 = 80
q selection
65
0
$M_{\{32(n-1)...32n-1\}}$*qi
Only for PS_0 and PS_1
Only connected to PS_2 + M_2qi
(4:2) Adder
Carry test
0
31
32
63
64
95
96
127
128
159

Figure 6.16 Radix-4 operation

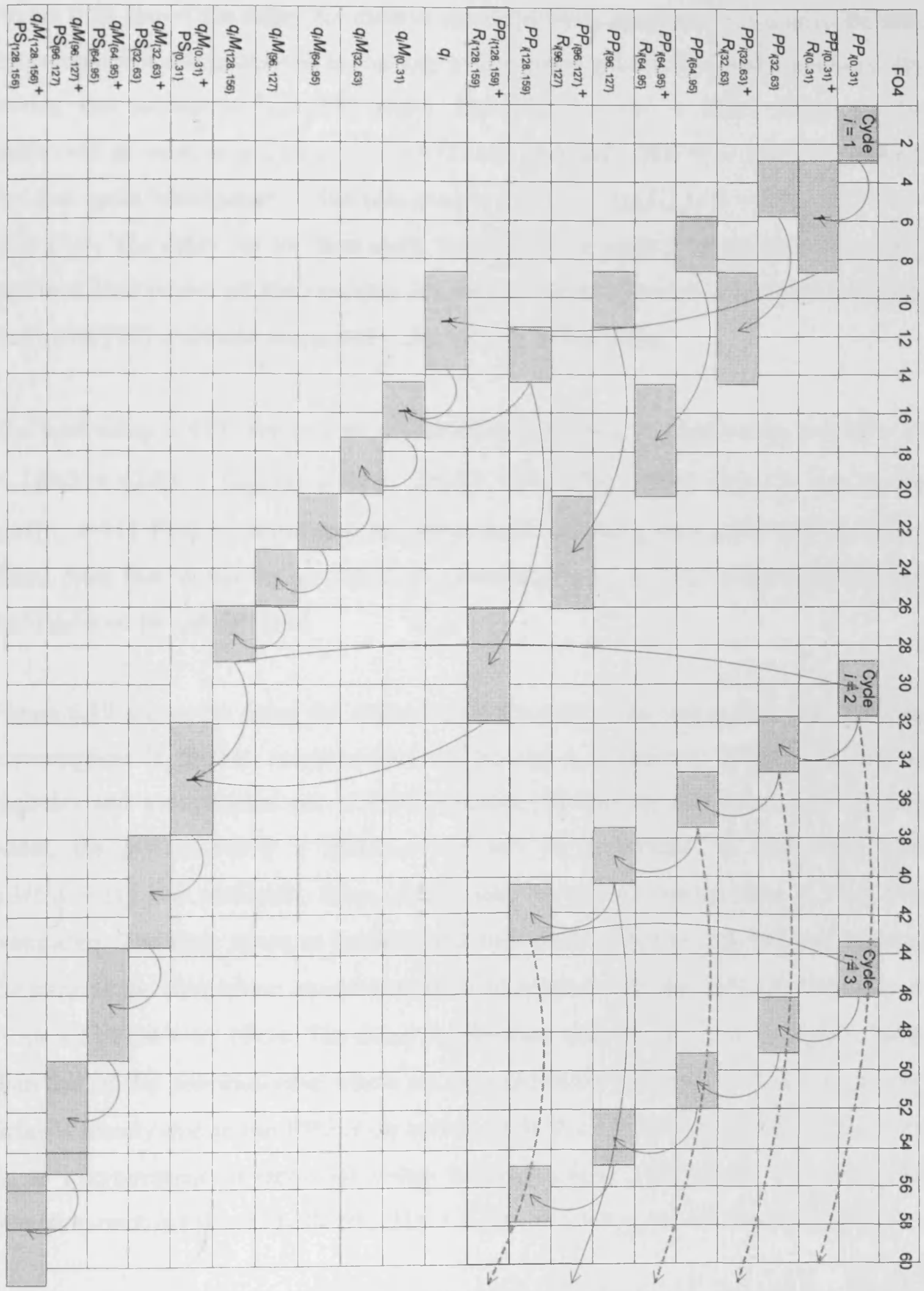


Figure 6.17 Timing diagram for radix-4 operation (reuse all modules)

Figure 6.18 shows the delay for radix-4 multiplication, however in this case the area consumption is compromised by having a duplicated set of (4:2) adder modules. By having this second set of (4:2) adder, the $q_iM\{i...i+31\}$ + PS$\{i...i+31\}$ can be preformed as soon as q_i and $q_iM\{i...i+31\}$ are available. The time taken to perform the first cycle Montgomery Multiplication is $[(2.85 + \log_4 32) \times 6 + 5.8 \times 5]$ FO4 = 61.1 FO4. The delay for the first cycle in this case is slightly faster then that of the previous case where all the modules are reused. This is because the delay is mainly due to the PPG $A \times bi$ and the $q_iM\{0...31\}$ and its buffer delay.

The total delay in FO4 for 160 bit × 160 bit Montgomery Multiplication is $[(5.8 \times 5) \times 160/2 + (2.85 + \log_4 32) \times 6]$ = 2352.1 FO4. The overall delay is due to the $q_iM\{i...i+31\}$ PPG modules and the PPG $A \times bi$ and the $q_iM\{0...31\}$ and its buffer delay. Note that by having an extra (4:2) module, the total delay of the operation has been reduced by almost 50%.

Figure 6.19 shows the delay for radix-4 multiplication, however in this case, the area consumption is further compromised by having a duplicated set of (4:2) adder modules and a duplicated set of PPG modules. By having this second set of (4:2) adder, the $q_iM\{i...i+31\}$ + PS$\{i...i+31\}$ can be preformed as soon as q_i and $q_iM\{i...i+31\}$ are available. Also, $A \times b_{i+1}$ can be carried out as soon as $A \times b_i$ has completed. The time taken to perform the first cycle Montgomery Multiplication is the same as the case where an extra set of (4:2) adder is included: $[(2.85 + \log_4 32) \times 6 + 5.8 \times 5]$ FO4 = 61.1FO4. The delay for the first cycle in this case is slightly faster than that of the previous case where all the modules are reused. This is because the delay is mainly due to the PPG $A \times bi$ and the $q_iM\{0...31\}$ and its buffer delay. There is no improvement in terms of delay because $q_iM\{0...31\}$ could only begin its operation once $b_iA\{0...31\}$, $PP\{0...31\} + R\{0...31\}$ and q_i selection are completed.

The total delay in FO4 for 160 bit × 160 bit Montgomery Multiplication is $[(5.8 \times 5) \times 160/2 + (2.85 + \log_4 32) \times 6]$ = 2352.1 FO4. The overall delay is due to the $q_iM\{i...i+31\}$ PPG modules and the PPG $A \times bi$ and the $q_iM\{0...31\}$ and its buffer delay. Note that by having an extra PPG unit, the total delay of the operation is not affected compared with the previous case. This is because by having extra PPG modules, more $A \times bi$ can be performed within a cycle, however as the critical path lies on $q_iM\{i...i+31\}$ + PS$\{i...i+31\}$, the improve performance on PPG does not affect the overall delay of the Montgomery Multiplication.

In conclusion, if the multiplier is designed with hardware area consumption in mind where there is only one of each module, the overall delay is the slowest. By having an extra set of (4:2) adders, the total delay is improved significantly and the extra area consumption is:

- 3×32 XOR/XNOR
- 2x32 NOR
- 1×32 NAND
- 1×32 NOT
- 2 ×32OAI CMOS complex gates
- 1×32 AOI CMOS complex gate

Finally, if both the adder and the PPG units are duplicated, it provides no extra benefits in terms of delay, but at the same time the extra PPG units consume the following:

- 2×32 AND
- 1×32 OR
- 1×32 XOR
- 2×32 4-input MUX

Therefore, having extra set of (4:2) redundant adders provides the best trade off between delay and area.

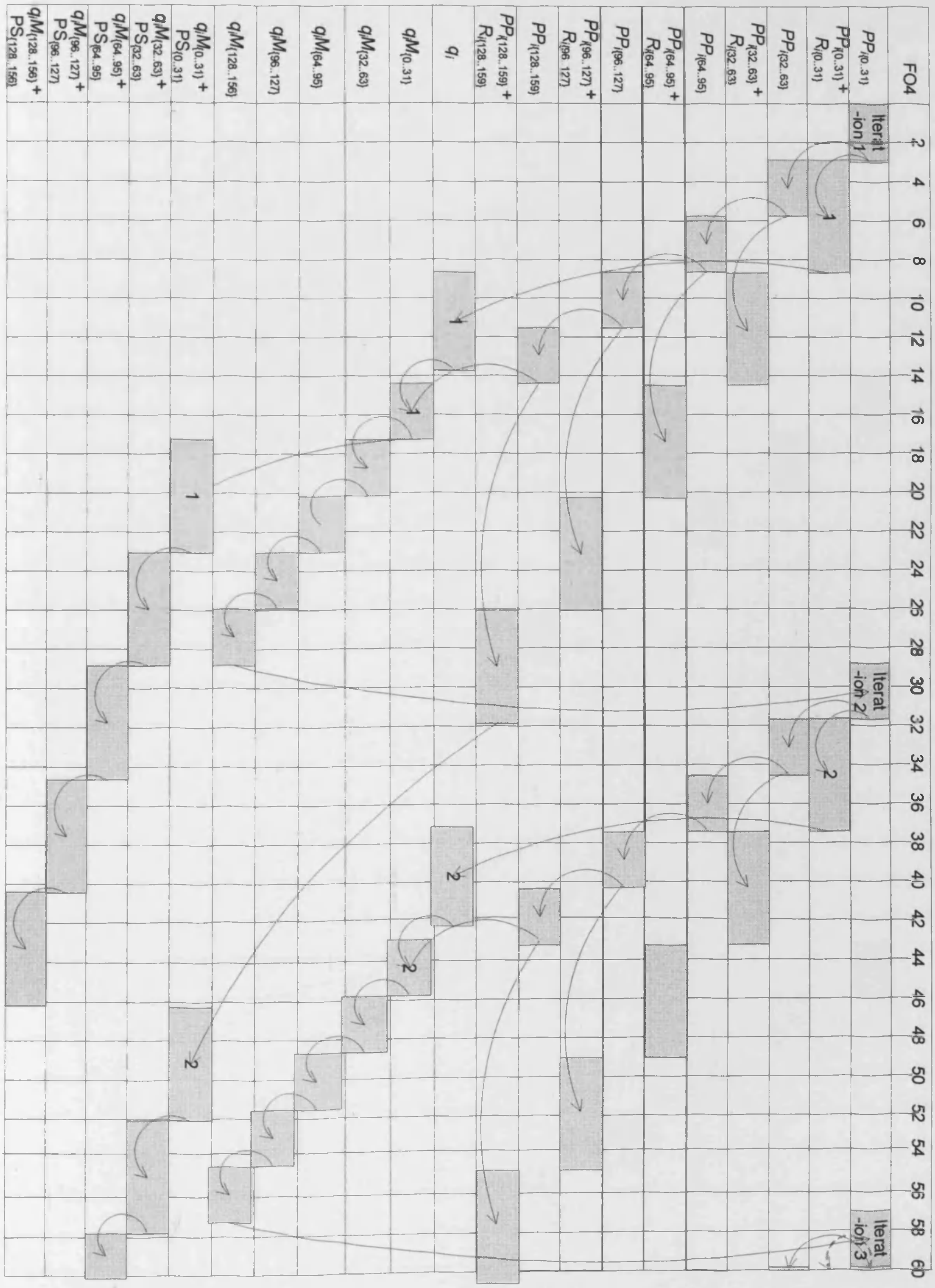


Figure 6.18 Radix-4 operation (duplicated adder modules)

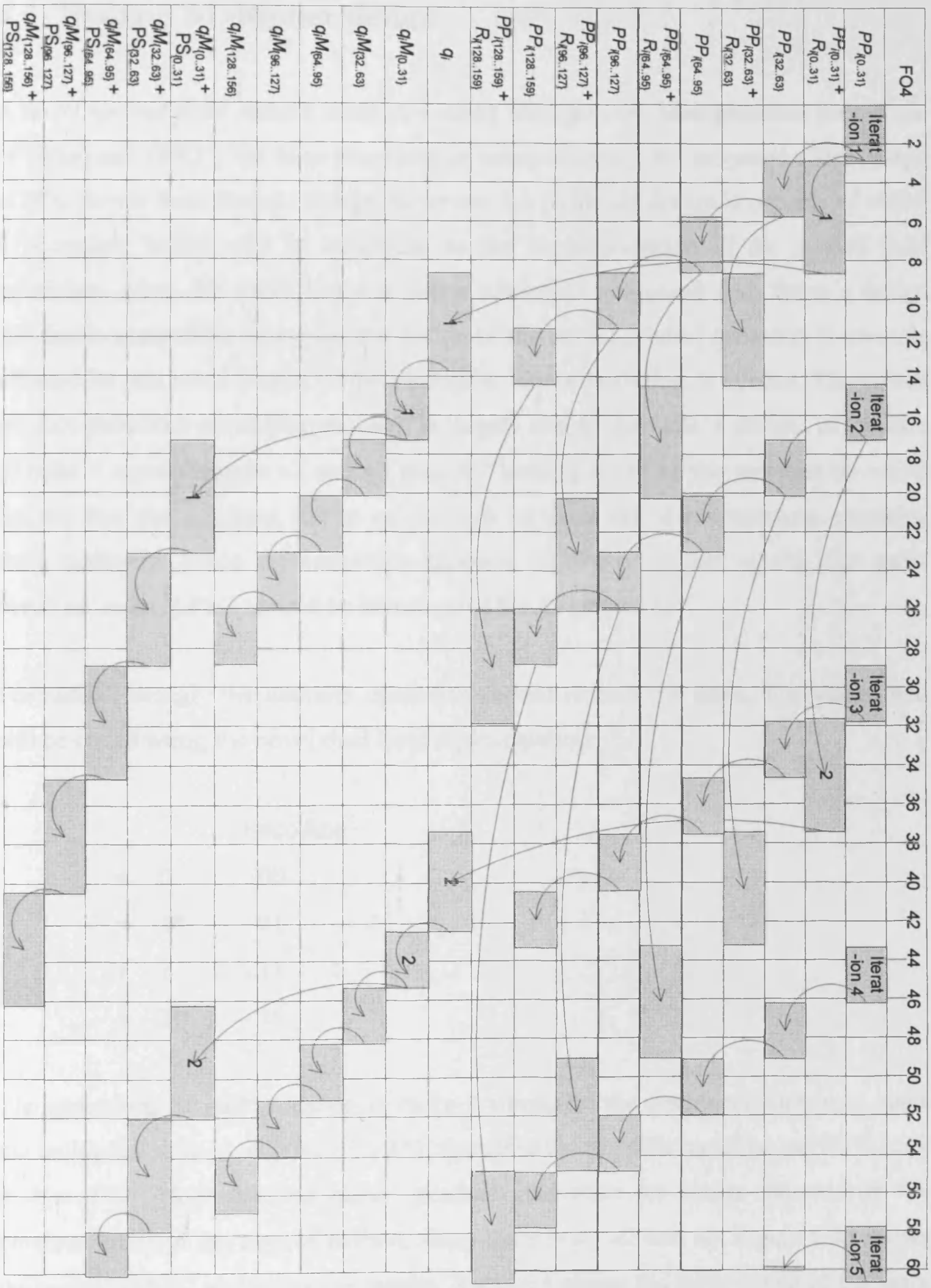

FO4
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
50
52
54
56
58
60
Iterat -ion 1
Iterat -ion 2
Iterat -ion 3
Iterat -ion 4
Iterat -ion 5
PP_{l(0..31)}
PP_{j(0..31)} + R_{l(0..31)}
PP_{l(32..63)}
PP_{j(32..63)} + R_{l(32..63)}
PP_{l(64..95)}
PP_{j(64..95)} + R_{l(64..95)}
PP_{l(96..127)}
PP_{j(96..127)} + R_{l(96..127)}
PP_{l(128..159)}
PP_{j(128..159)} + R_{l(128..159)}
q_i
q_iM_{(0..31)}
q_iM_{(32..63)}
q_iM_{(64..95)}
q_iM_{(96..127)}
q_iM_{(128..156)}
q_jM_{(0..31)} + PS_{(0..31)}
q_jM_{(32..63)} + PS_{(32..63)}
q_jM_{(64..95)} + PS_{(64..95)}
q_jM_{(96..127)} + PS_{(96..127)}
q_jM_{(128..156)} + PS_{(128..156)}
1
2

Figure 6.19 Radix-4 operation (duplicated adder and PPG modules)

6.4 Radix-2 Multiplier design

A novel unified field radix-4 multiplier using Montgomery Multiplication for the use of GF(p) and GF(2^n) has been proposed. In terms of delay, the proposed adder design is 24% slower than Savaş's design, however, the proposed design is capable of radix-4 operation, which will be beneficial to the implementation of the unified field multiplier. Also, the adder has one major advantage compared with Savaş's design and that is scalability. However, the design of the partial product generator is severely affected by the word length of the multiplier where buffering is needed. The partial product generator scalability problem is largely due to the radix 4 design, in the case of radix 4 multiplication ×2 and ×3 require "looking back" to the previous bit which implies that the q_i signal has to go through an extra set of multiplexers, requiring extra buffering. Since the scalability problem of PPG is caused by the high radix involved, radix 2 PPG should be investigated for its suitability.

For radix-2 design, the addition operation should remain the same. The values will still be coded using the novel dual field representation:

	Encoding
• 0	- 00
• 1*	- 01
• 1	- 11
• 2	- 10

The operations of multiplication in radix-2 simplifies the circuit requirements, since the multiplicand b_i is only 0 or 1 (1*), therefore the possible outcome per bit is only 0, 1(or 1*). Note that the partial product generation no longer depends on the previous bit as in the case of radix-4, since there is no ×2 and ×3. Figure 6.20 shows the overall radix 2 multiplication results. Table 6.6 shows the logic circuit of the radix 2 PPG unit.

Table 6.6 Radix 2 multiplication

A_i, A_{i-1}	$b_i = 0$	$b_i = 1$
00	00	00
01	00	01
10	00	10

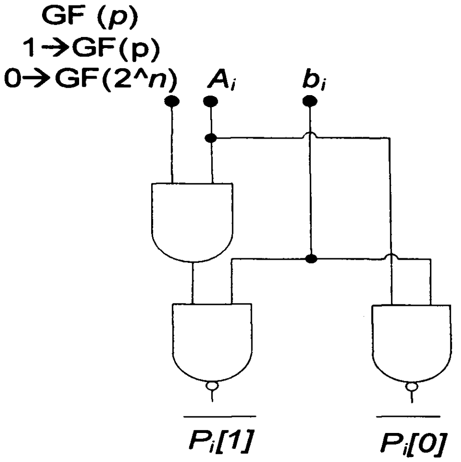


Figure 6.20 Radix 2 PPG unit

6.4.1 Q_i-selection

In order to carry out Montgomery reduction, an appropriate multiple of the modulus must be added to the Partial Sum *PS*, so that the last bit will equal 0. Unlike the process of Montgomery reduction in radix–4, only one bit is under consideration in the radix-2 case, which simplifies the matter greatly. For both GF(p) and GF(2^n), 0+0 = 0 and 1+1 = 0. Note that since the LSB of M is always 1, therefore $q_i = PS_0[0]$. Hence, no delay is due to the logic; however, q_i signal has high propagation delay as discussed earlier in this chapter. Table 6.7 shows all radix 2 q-selection possibilities.

Table 6.7 Radix 2 q- selection

$PS_0 + q_iM_0$	***q_iM_0 = 0* 1***	***q_iM_0 = 1* 1***
00 (0)	00	-
01 (1*)	-	00
10 (2)	00	-
11 (1)	-	00

6.4.2 Carry Test

Table 6.8 All possible PS

PS	*R*									
		00	01	02	10	11	12	20	21	22
b_iA	00	00	01	10	10	11	20	00	01	10
	01	01	10	11	11	20	01	01	10	11
	10	10	11	20	00	01	10	10	11	20

Table 6.9 *PS* and *q_iM* combination

+	*$(PS_1)PS_0$*					
		(0)0	(0)1	(1)0	(1)1	(2)0
q_iM_0	+0	00	-	0	-	0
	+1	-	10 (c = 1)	-	0 (c = 1)	-

Table 6.8 shows that the last bit of PS_0 will never become 2, however, a carry will be propagated to the next bit when PS_0 is 1 or 2 (GF(p) only). Therefore, only $PS_0[1]$ needs to be examined, where $PS_0[1] = 0$, carry test also is 0 and when $PS_0[1] = 1$, carry test also is 1. This is possible because $PS_0 = 2$ would not happen and in the case of GF(2^n), $PS_0[1] = 0$. Hence, the carry test required is identical to the value of $PS_0[1]$ as summarised in Table 6.9.

6.4.3 q_iM + PS

In the radix-4 case (see Figure 5.24), it has already been shown that the 2 LSB (4:2) dual field redundant adders are no longer necessary, in the case of radix-2, the LSB is no longer needed. It was mentioned before that for the partial product generation, since only ×0 and ×1 are required, the PPG modules are not dependent on the previous bit; therefore q_iM_0 serves no purpose in this design. Figure 6.21 shows the overall simplified radix-2 dual field Montgomery multiplier architecture.

6.4.4 M-bit × M-bit multiplication using radix-2

As in the radix-4 multiplication case (see section 6.3), the operation of a 160-bit × 160-bit multiplication radix-2 multiplication, where the size of the multiplier is 32-bit is shown as follows:

For radix-2, the partial product generation $A*B$ will be done in 160/32 * 160/1 cycles = 800 cycles. This is double the number of cycle required by radix-4 multiplier. Since multiplier A is multiplied by only 1 bit of b_i at each cycle. Note that the register size required for $A_{\{n...n+31\}}*b_i$ is 32*2 = 64 bit instead of 66 bit as in the radix-4 case because A is now multiplied by either 0 or 1 . Apart from the points mentioned above, the structure and operation of the radix-2 multiplier is the same as the proposed radix-4 version. The overall radix-2 operation is show in Figure 6.22, it shows that the q_iM should only be computed only once the computation of $A*bi$ is completed. This is to avoid the need of having to have two sets of partial product generation modules.

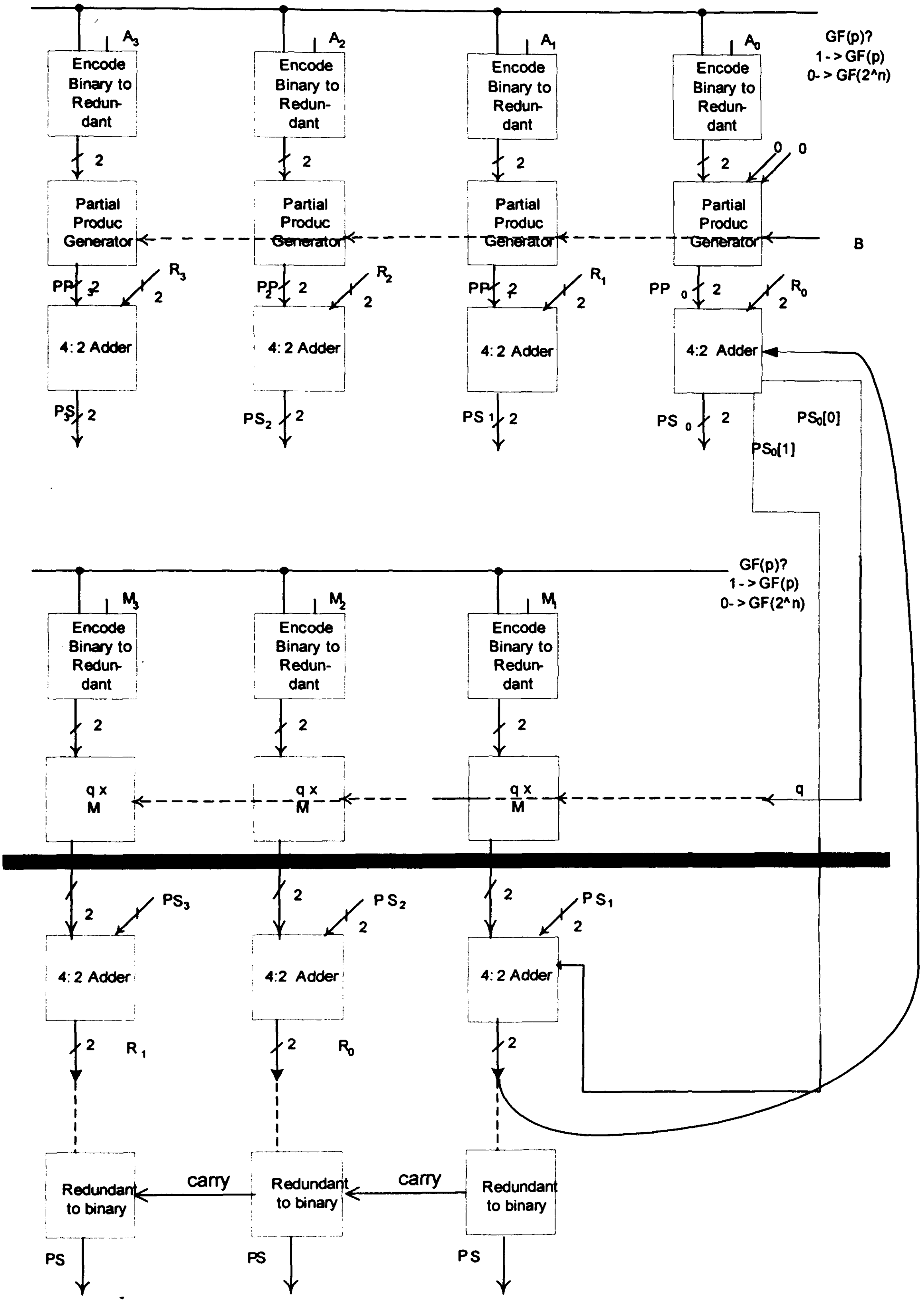


Figure 6.21 Radix 2 Overall Architecture

The delay of the radix-2 multiplication is found as follows:

The input load is connected to w-bit of PPG modules and each module consists of two 2-input NAND gates. Therefore, the branching effort (b) for b_i (including allowance for wires) is $(2\times4/3 + 2\times2/3)w = 4w$. Table 6.10 shows the logical effort calculation of the radix 2 Montgomery Multiplier.

Table 6.10 Radix -2 Redundant Montgomery Multiplier Delay

	Logical effort *g*	**Branching effort *b***	**Electrical effort *h***	**Parasitic effort *p***	**Path effort *gb(h)***
Input B[1]	1	$(2 \times 4/3 + 4/3)\ w\ /\ 4/3$	1	-	$3w$
NAND	2	(22/3) / 4	1	2	22/6

Path Effort of the critical path $F = GBH$ (critical path) = $11w$

The number of stages N needed including buffers can be calculated as follows:

$$N = \text{rnd}\,(\log_4 F)$$

$$= \log_4 11 = \ln(11)/\ln 4 = 1.73$$

$$\text{Rnd } 1.73 = 2$$

$$\therefore N = 2$$

Therefore 2 stages are needed, i.e. one buffer stage is added.

$$\alpha = F^{1/N} = 11^{1/2} = 3.32$$

$$D = (N \times \alpha + P)/5 = (2\times3.32 + 2+1)/5 = 1.93 \text{ FO4}$$

Figure 6.22 shows the overall operation of the 160×160 bit radix 2 multiplier. Figure 6.23 shows that the addition takes up 5.8 FO4, whereas partial product generation using a 2-input NAND gate is only 1.93 FO4.

Hence, the partial product generator finishes all the calculations for the i^{th} iteration before the (4:2) adder even completes the first half of addition. Partial product generation for the next iteration i+1 ($A*b_{i+1}$), i+2 ($A*b_{i+2}$), i+3 ($A*b_{i+3}$) and i+4 ($A*b_{i+4}$) can be performed before the adder is ready to compute addition for the next iteration.

Radix-2 design reduces the delay of partial product generation, Montgomery modulus selection which caused scalability problem and caused irregularity to the design and affect the critical path in the radix-4 case. The scalability of the radix-4 design is proven as follows:

It has been worked out that the (4:2) adder has a delay of 5.8 FO4, it has been mentioned that since the carry in to the adder is 0, which means that the LSB is ready after only one XOR gate effectively $\approx 5.8/3 = 1.93$ FO4. The delay of the q_iM is also 1.93 FO4. Therefore, the time allowed for buffer delay where the (4:2) adder delay is 5.8 FO4 is calculated as: (5.8 – 1.93) FO4 to buffer q_i ($\log_4 w$) and form q_iM (1.93). Therefore, even without quotient pipelining can have w=16 radix-2 modulo multiplier adder whose delay is not impacted by w.

If quotient pipelining is applied, similar calculations can be carried out except there are 5.8 FO4 to buffer q_i ($\log_4 w$) and form q_iM (1.93), instead of (5.8 – 1.93) as in the previous case. Therefore, $w = 4^{3.9} = 222$ is allowed without impacting the delay, hence, radix-2 design is scalable especially when quotient pipelining is employed.

However, as can be seen in Figure 6.23, the delay of the addition dominates the critical path, therefore the reduction in speed in the PPG operations and Montgomery modulus selection does not affect the overall delay. Since for radix-2 operations, twice as many iterations as needed for radix-4 are required; hence, although radix-2 operation has better scalability, its delay is two times worse than the radix-4 design.

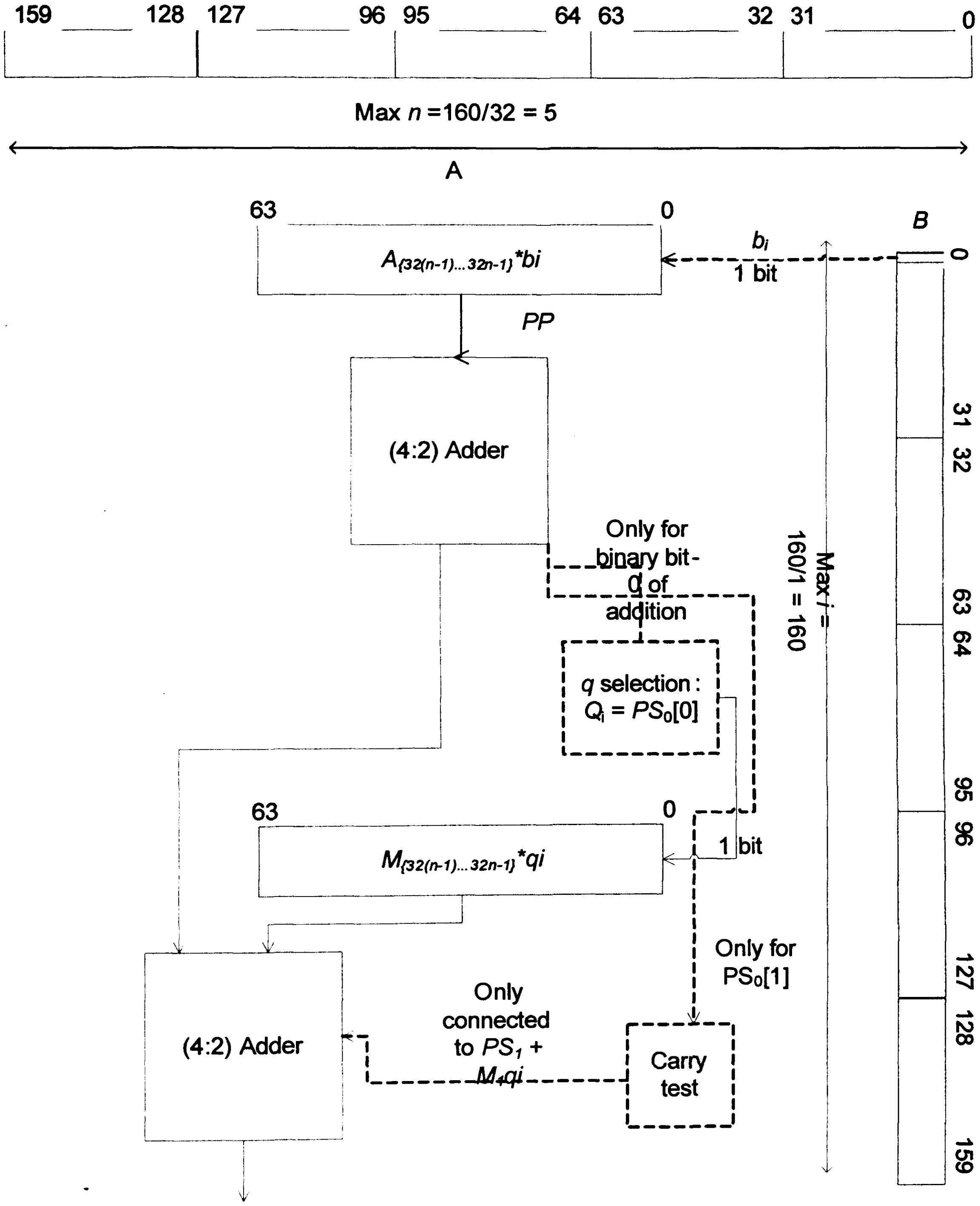


Figure 6.22 Radix 2 operation

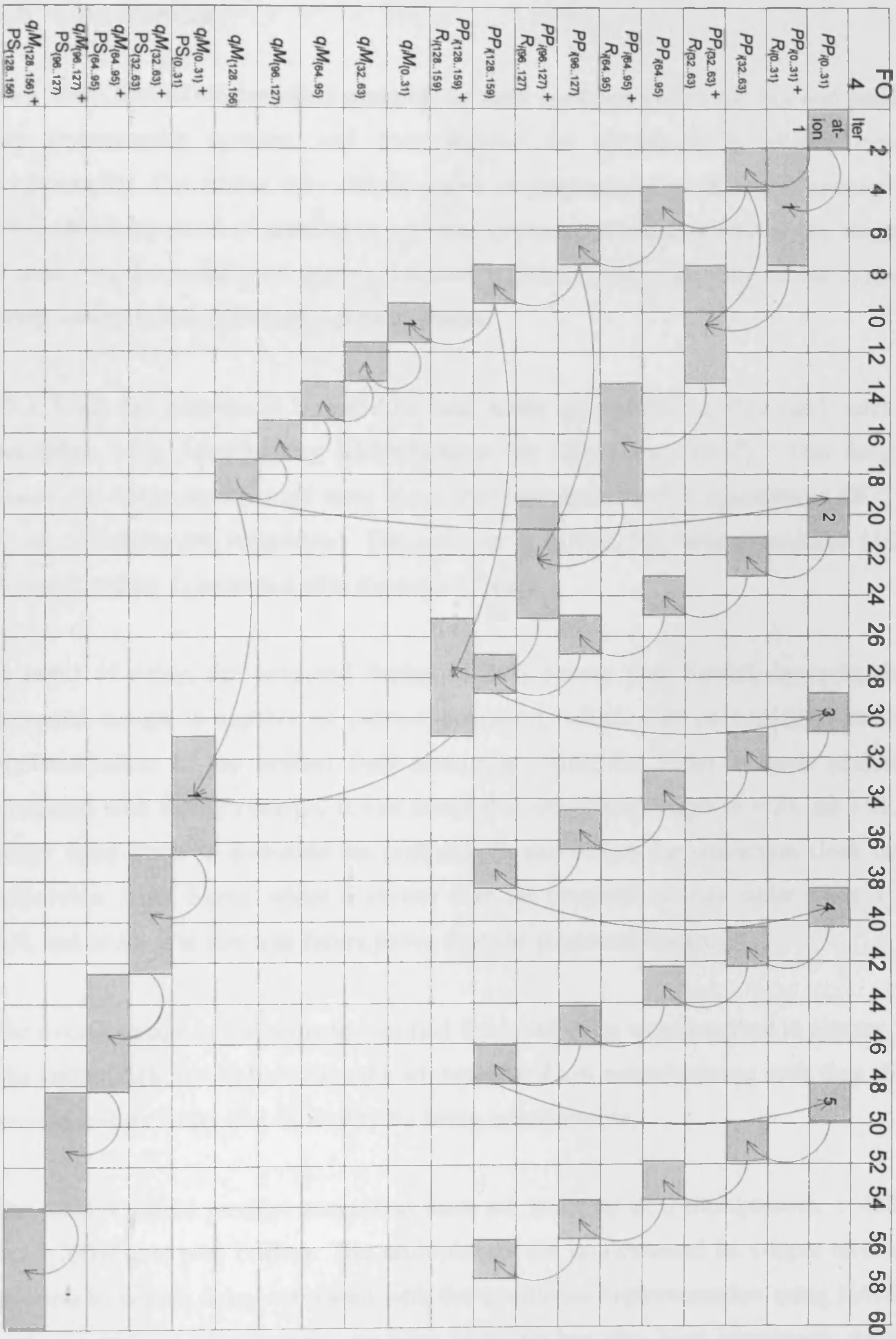


Figure 6.23 Radix 2 (reuse all modules)

6.5 Conclusion

This thesis first of all described some of the most common symmetric key and public key cryptography systems, and then showed the operations of elliptic curve cryptography. The reason why elliptic curve cryptography was chosen is because it provides similar level of security as previous systems but requires smaller key length. It improves the security of system, because it reduces the possibility of the system being susceptible to differential power attacks.

This thesis has introduced a new dual-field adder and a novel unified field radix-4 multiplier using Montgomery Multiplication for GF(p) and GF(2^n). This design makes use of the unexploited state in number representation for operation in GF(2^n) where all carries are suppressed. The addition is carried out using a modified (4:2) redundant adder to accommodate the extra 1* state.

In terms of delay, the proposed design is 24% slower than Savas', however, the proposed design is capable of radix-4 operation, which will be beneficial to the implementation of the unified field multiplier. Also, the adder is more scalable compared with Savaş's design. It was found that when wordlength $w = 23$, the FSEL buffer delay starts to dominate the critical path and affects the maximum clock rate achievable. Also, Savaş' adder is slower than the proposed unified adder when $w >$ 109, and hence it is also less future prove than the proposed design.

The overall design of the proposed unified field multiplier was described in chapter 5. The unified field multipliers have the advantages of low manufacturing cost, they also provide compatibility and flexibility by being interoperable.

The radix-4 partial product generation units are made up of 2 multiplexers, 1 AND and 1 XOR gate plus buffers. The multiplexers are implemented as 4-input tri-state inverters to reduce delay compared with the traditional implementation using NAND gates. A radix-4 Montgomery modulus selection has also been introduced, which comprises 1 multiplexer and 2 XOR gates. It was also shown that 4-way inverting multiplexer significantly improve the delay and scalability of the multiplier compared with traditional multiplexer design.

The proposed Montgomery multiplier possesses some unique features such as the use of the 1* encoding, however, it does not provide the expected degree of improvement over the previously proposed design. It has been identified that the partial product generator modules is one of the main causes of poor scalability. The combined delay of the partial product generator and multiplier digit selector for w bits is $2.85 + \log_4 w + 5.01 = \log_4 w + 7.86$. This delay is greater than the (4:2) adder and so adequate pipelining methods could be used to minimise the impact. Quotient pipelining was discussed as a possible pipelined architecture.

M-bit × M-bit multiplication operation was investigated for both radix-4 and radix-2 design. Three different scenarios were assessed: (1) both the adders and PPG modules are reused; (2) reuse PPG modules only and (3) reuse both adder and PPG modules. The best trade off between speed and area consumption is the second case where only PPG modules are reused.

Radix-2 PPG design reduces the delay, area and the scalability requirement of the multiplier circuit; however, it increases the overall delay significantly due to doubling the number of cycles required. Radix-4 design has limited scalability due to the buffering delay of q_i, but the overall delay is better than that of radix-2 design, especially in the case when a separate adder is available.

Further investigation in the design of the multiplier could be done by exploring different PPG designs. It is understood that increasing the radix will not improve the situation since the circuit design of the PPG modules will become very complex to perform ×4, ×5, ×6 and ×7. Another possibility is to explore the idea of mixed radix architecture, with higher radix for GF(2^n), since the GF(2^n) design is simpler because of the lack of carry propagation, however, this will favour the theoretically less complex design of GF(2^n), i.e there is no longer impartial between fields.

Appendix 1 – Algorithms

1. Extended Euclidean Algorithm [157]

Given nonnegative integers u and v, this algorithm determines a vector ($u1$, $u2$, $u3$) such that $uu1 + vu2 = u3 = \gcd(u, v)$. The computation makes use of auxiliary vectors ($v1$, $v2$, $v3$), ($t1$, $t2$, $t3$); all vectors are manipulated in such a way that the relations

$$ut_1 + vt_2 = t_3,\ uu_1 + vu_2 = u_3,\ uv_1 + vv_2 = v_3$$

Hold through the calculation

*X*1. [Initialize.] Set $(u_1, u_2, u_3\} \leftarrow (1, 0, u)$, $(v_1, v_2, v_3) \leftarrow (0, 1, v)$

*X*2. [Is $v_3 = 0$?] If $v3 = 0$, the algorithm terminates.

*X*3. [Divide, subtract.] Set $q \leftarrow \lfloor u_3/ v_3 \rfloor$ and then set

$$(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - (v_1, v_2, v_3)q,$$

$$(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3), \qquad (v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3)$$

Return Step *X*2.

2. Repeated square-and-multiply algorithm for exponentiation in $\mathbf{Z_n}$ [158]

INPUT: $a \in Z_n$ and integer $0 \leq k < n$ whose binary representation is $k = \sum_{i=0}^{t} k_i 2^i$.

OUTPUT: $a^k \bmod n$

```
1. Set $b \leftarrow 1$. If $k = 0$ then return ($b$).
2. Set $A \leftarrow a$.
3. If $k_0 = 1$ then set $b \leftarrow a$.
4. From $i$ from 1 to $t$ do the following:
       1.   Set $A \leftarrow A^2$ mod n
       2.   If $k_i = 1$ then set $b \leftarrow A \bullet b \bmod n$
5. Return ($b$).
```

3. Chinese Remainder Theorem (CRT)

Let $m_1, m_2, \ldots, m_r$ be positive integers that are relatively prime in pairs, i.e.,

$$\gcd(m_j, m_k) = 1 \qquad \text{when } j \neq k.$$

Let $m = m_1, m_2, \ldots m_r$, and let $a, u_1, u_2, \ldots, u_r$ be integers. Then there is exactly one integer u that satisfies the conditions:

$$a \leq u < a + m$$

and

$$u \equiv u_j (\text{modulo } m_j) \qquad \text{for } 1 \leq j \leq r.$$

4. Fermat's Little Theorem

If p is a prime and a is an integer with $\gcd(a, p) = 1$, then

$$a^{p-1} \equiv 1 (\bmod p)$$

Appendix 2 – Logical Effort

Logical effort was first introduced by [131] and [159] describes logical effort in details. Logical effort is a design model to estimate the performance of CMOS logical circuit, namely the number of CMOS stage including buffers required and the overall delay of the circuit.

The delay of CMOS logic gate (*d*) is defined as:

$$d = f + p \qquad \text{(A-2. 1)}$$

Where *f* denotes effort delay and *p* denotes parasitic delay

The effort delay (*f*) consists of two components:

$$f = gh \qquad \text{(A-2. 2)}$$

Where *g* denotes logical effort and *h* denotes electrical effort. Therefore,

$$d = gh + p \qquad \text{(A-2. 3)}$$

This is equivalent to:

$$d = load\, /\, drive + t_{gate} \qquad \text{(A-2. 4)}$$

Definitions:

1. Logical effort *g* – the input capacitance of a logic gate relative to that of a minimum size inverter
2. Electrical effort *h* – ratio of output capacitance to gate input capacitance

3. Parasitic delay p – total diffusion capacitance on the output node of a CMOS logic gate relative to the input FET gate capacitance of a minimum-sized inverter

Thus, Table A-2. 1 and Table A-2. 2 can be formed based on the definitions shown above.

Table A-2. 1 Logical effort of static CMOS gates

Gate	Number of inputs					
	1	2	3	4	5	n
Inverter	1					
NAND		4/3	5/3	6/3	7/3	$(n+2)/3$
NOR		5/3	7/3	9/3	11/3	$(2n+1)/3$
Multiplexer		2	2	5	2	2
XOR, XNOR		4	12	32		

Table A-2. 2 Parasitic delay of static CMOS gates

Gate	Parasitic Delay
Inverter	$P_{inv} = 1$
n-input NAND	nP_{inv}
n-input NOR	nP_{inv}
n-way multiplexer	$2nP_{inv}$
2-input XOR, XNOR	$4nP_{inv}$

It is common to approximate the delay in terms of fan-out = 4 ("FO4") inverter delays. FO4 delay means that the delay of 1 NOT gate with load of 4 NOT gates:

$$d = gh + p \quad \text{(A-2. 5)}$$

$$d_{FO4} = 1 * 4 + 1 = 5 \quad \text{(A-2. 6)}$$

Therefore, to get the FO4 delay, simply divide the total gate delay by 5.

Logical effort can also estimate the delay along the critical path; however, Branching effort *b* and Path effort *F* will also need to be considered.

Definitions:

4. Branching effort *b* - ratio of total capacitative load on one CMOS logic gate's output along the critical path to the FET gate capacitance of the next CMOS gate on the critical path

5. Path effort *F*:

   $$F = GBH \text{ (critical path)} \quad \text{(A-2. 7)}$$

   Where $G = \Pi g$, $B = \Pi b$, and $H = \Pi h$.

The total electrical effort *H* reduces to the ratio of the output capacitance loading the last CMOS logic gate to the FET gate capacitance of the first CMOS logic gate along the critical path (C_{out}/ C_{in}). Usually, *H* is forced to 1 by assuming that the circuit being modelled is connected to a copy of itself. This allows input branching effort to be incorporated in a delay estimate, and allows individual subcircuits to be modelled independently before being cascaded to form a whole.

6. Delay along a critical path D:

$$\begin{aligned} &D = N\alpha + P \\ &or \\ &D = N\alpha + P/5(FO4) \end{aligned} \qquad \text{(A-2. 8)}$$

Where N is the number of CMOS gates (including buffer) and α is load/drive.

7. The number of stages (including buffer) N can be found by:

$$N = \text{rnd}\ (\log_{3.6} F) \qquad \text{(A-2. 9)}$$

Where "rnd (x)" denotes round up or down to the nearest integer.

8. load/drive α is defined as:

$$\alpha = F^{1/N} \qquad \text{(A-2. 10)}$$

The delay of the critical path as shown in Equation A-2.8 can now be calculated.

Appendix 3 - Synthesis result report

```
Information: Updating design information... (UID-85)

****************************************
Report : timing
        -path full
        -delay max
        -nworst 10
        -input_pins
        -nets
        -max_paths 100
        -capacitance
Design : top_level
Version: 2000.11-SP1
Date   : Fri Aug  1 14:59:33 2003
****************************************

Operating Conditions: tsmc18os120_max   Library: tsmc18os120_max
Wire Load Model Mode: enclosed

  Startpoint: uClock_Timer/y_reg[0]
              (rising edge-triggered flip-flop clocked by CLK)
  Endpoint: T_S[2] (output port clocked by CLK)
  Path Group: CLK
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  top_level          4000                  tsmc18os120_max
  Clock_Timer        4000                  tsmc18os120_max
  sixteenFA_1        ForQA                 tsmc18os120_max
  Four_mux_6         ForQA                 tsmc18os120_max
  CELL_C_6           ForQA                 tsmc18os120_max
  mux                ForQA                 tsmc18os120_max
  sixteenFA_0        4000                  tsmc18os120_max
  CELL_A_3           ForQA                 tsmc18os120_max
  eight_two          4000                  tsmc18os120_max
  CELL_B_5           ForQA                 tsmc18os120_max
  q_selector         ForQA                 tsmc18os120_max

  Point                                       Fanout      Cap     Incr      Path
  -----------------------------------------------------------------------------------
  clock CLK (rise edge)                                           0.00      0.00
  clock network delay (ideal)                                     0.00      0.00
  uClock_Timer/y_reg[0]/CP (dfcrn1)                               0.00      0.00 r
  uClock_Timer/y_reg[0]/QN (dfcrn1)                               0.31      0.31 f
  uClock_Timer/n203 (net)                       1         0.01      0.00      0.31 f
  uClock_Timer/U59/I (inv0d1)                                     0.00      0.31 f
  uClock_Timer/U59/ZN (inv0d1)                                    0.10      0.41 r
  uClock_Timer/y[0] (net)                       2         0.01      0.00      0.41 r
  uClock_Timer/y[0] (Clock_Timer)                                 0.00      0.41 r
  y[0] (net)                                              0.01      0.00      0.41 r
  ueight_two/y[0] (eight_two)                                     0.00      0.41 r
  ueight_two/y[0] (net)                                   0.01      0.00      0.41 r
  ueight_two/U1/y[0] (sixteenFA_1)                                0.00      0.41 r
  ueight_two/U1/y[0] (net)                                0.01      0.00      0.41 r
  ueight_two/U1/A1/y_0 (CELL_A_7)                                 0.00      0.41 r
  ueight_two/U1/A1/y_0 (net)                              0.01      0.00      0.41 r
  ueight_two/U1/A1/U13/A2 (xr02d2)                                0.00      0.41 r
  ueight_two/U1/A1/U13/Z (xr02d2)                                 0.42      0.83 f
  ueight_two/U1/A1/S_a_0 (net)                  3         0.03      0.00      0.83 f
  ueight_two/U1/A1/S_a_0 (CELL_A_7)                               0.00      0.83 f
  ueight_two/U1/S_a[0] (net)                              0.03      0.00      0.83 f
  ueight_two/U1/B1/S_a_0 (CELL_B_5)                               0.00      0.83 f
  ueight_two/U1/B1/S_a_0 (net)                            0.03      0.00      0.83 f
  ueight_two/U1/B1/U13/A2 (nd12d2)                                0.00      0.84 f
  ueight_two/U1/B1/U13/ZN (nd12d2)                                0.07      0.90 r
  ueight_two/U1/B1/n107 (net)                   1         0.01      0.00      0.90 r
  ueight_two/U1/B1/U12/A1 (an02d1)                                0.00      0.90 r
  ueight_two/U1/B1/U12/Z (an02d1)                                 0.19      1.09 r
  ueight_two/U1/B1/C_b (net)                    1         0.01      0.00      1.09 r
  ueight_two/U1/B1/C_b (CELL_B_5)                                 0.00      1.09 r
  ueight_two/U1/C_b[0] (net)                              0.01      0.00      1.09 r
  ueight_two/U1/C2/C_b (CELL_C_6)                                 0.00      1.09 r
```

```
ueight_two/U1/C2/C_b (net)                          0.01      0.00      1.09 r
ueight_two/U1/C2/U9/I (inv0d2)                                0.00      1.09 r
ueight_two/U1/C2/U9/ZN (inv0d2)                               0.05      1.14 f
ueight_two/U1/C2/n103 (net)                  2      0.02      0.00      1.14 f
ueight_two/U1/C2/U8/A2 (xn02d2)                               0.00      1.14 f
ueight_two/U1/C2/U8/ZN (xn02d2)                               0.34      1.48 f
ueight_two/U1/C2/S_c_0 (net)                 3      0.02      0.00      1.48 f
ueight_two/U1/C2/S_c_0 (CELL_C_6)                             0.00      1.48 f
ueight_two/U1/S[2] (net)                            0.02      0.00      1.48 f
ueight_two/U1/S[2] (sixteenFA_1)                              0.00      1.48 f
ueight_two/aS[2] (net)                              0.02      0.00      1.48 f
ueight_two/q/x1_0 (q_selector)                                0.00      1.48 f
ueight_two/q/x1_0 (net)                             0.02      0.00      1.48 f
ueight_two/q/U8/A2 (xr02d1)                                   0.00      1.48 f
ueight_two/q/U8/Z (xr02d1)                                    0.32      1.81 f
ueight_two/q/wire_b (net)                    1      0.01      0.00      1.81 f
ueight_two/q/mux1/b (mux)                                     0.00      1.81 f
ueight_two/q/mux1/b (net)                           0.01      0.00      1.81 f
ueight_two/q/mux1/U11/I0 (mx02d2)                             0.00      1.81 f
ueight_two/q/mux1/U11/Z (mx02d2)                              0.31      2.11 f
ueight_two/q/mux1/n25 (net)                  1      0.03      0.00      2.11 f
ueight_two/q/mux1/U10/I (inv0d7)                              0.00      2.11 f
ueight_two/q/mux1/U10/ZN (inv0d7)                             0.06      2.18 r
ueight_two/q/mux1/n26 (net)                  1      0.04      0.00      2.18 r
ueight_two/q/mux1/U12/I (inv0da)                              0.00      2.18 r
ueight_two/q/mux1/U12/ZN (inv0da)                             0.06      2.23 f
ueight_two/q/mux1/op (net)                  13      0.16      0.00      2.23 f
ueight_two/q/mux1/op (mux)                                    0.00      2.23 f
ueight_two/q/q_1 (net)                              0.16      0.00      2.23 f
ueight_two/q/q_1 (q_selector)                                 0.00      2.23 f
ueight_two/q_1 (net)                                0.16      0.00      2.23 f
ueight_two/qM1/B[1] (qM_3)                                    0.00      2.23 f
ueight_two/qM1/B[1] (net)                           0.16      0.00      2.23 f
ueight_two/qM1/mux2/sel[1] (Four_mux_6)                       0.00      2.23 f
ueight_two/qM1/mux2/sel[1] (net)                    0.16      0.00      2.23 f
ueight_two/qM1/mux2/U12/S (mx02d0)                            0.02      2.25 f
ueight_two/qM1/mux2/U12/Z (mx02d0)                            0.32      2.58 r
ueight_two/qM1/mux2/n97 (net)                1      0.01      0.00      2.58 r
ueight_two/qM1/mux2/U10/I1 (mx02d1)                           0.00      2.58 r
ueight_two/qM1/mux2/U10/Z (mx02d1)                            0.21      2.79 r
ueight_two/qM1/mux2/op (net)                 2      0.02      0.00      2.79 r
ueight_two/qM1/mux2/op (Four_mux_6)                           0.00      2.79 r
ueight_two/qM1/P_0 (net)                            0.02      0.00      2.79 r
ueight_two/qM1/P_0 (qM_3)                                     0.00      2.79 r
ueight_two/P[0] (net)                               0.02      0.00      2.79 r
ueight_two/U2/y[0] (sixteenFA_0)                              0.00      2.79 r
ueight_two/U2/y[0] (net)                            0.02      0.00      2.79 r
ueight_two/U2/A1/y_0 (CELL_A_3)                               0.00      2.79 r
ueight_two/U2/A1/y_0 (net)                          0.02      0.00      2.79 r
ueight_two/U2/A1/U10/A1 (nd02d1)                              0.00      2.79 r
ueight_two/U2/A1/U10/ZN (nd02d1)                              0.11      2.89 f
ueight_two/U2/A1/n85 (net)                   1      0.01      0.00      2.89 f
ueight_two/U2/A1/U8/A1 (an02d4)                               0.00      2.89 f
ueight_two/U2/A1/U8/Z (an02d4)                                0.20      3.10 f
ueight_two/U2/A1/C_a (net)                   2      0.02      0.00      3.10 f
ueight_two/U2/A1/C_a (CELL_A_3)                               0.00      3.10 f
ueight_two/U2/C_a[0] (net)                          0.02      0.00      3.10 f
ueight_two/U2/B2/C_a (CELL_B_3)                               0.00      3.10 f
ueight_two/U2/B2/C_a (net)                          0.02      0.00      3.10 f
ueight_two/U2/B2/U10/A2 (xr02d2)                              0.00      3.10 f
ueight_two/U2/B2/U10/Z (xr02d2)                               0.39      3.49 f
ueight_two/U2/B2/S_b_0 (net)                 2      0.02      0.00      3.49 f
ueight_two/U2/B2/S_b_0 (CELL_B_3)                             0.00      3.49 f
ueight_two/U2/S_b[2] (net)                          0.02      0.00      3.49 f
ueight_two/U2/C2/S_b_0 (CELL_C_1)                             0.00      3.49 f
ueight_two/U2/C2/S_b_0 (net)                        0.02      0.00      3.49 f
ueight_two/U2/C2/U12/A1 (xr02d1)                              0.00      3.49 f
ueight_two/U2/C2/U12/Z (xr02d1)                               0.29      3.77 r
ueight_two/U2/C2/S_c_0 (net)                 1      0.00      0.00      3.77 r
ueight_two/U2/C2/S_c_0 (CELL_C_1)                             0.00      3.77 r
ueight_two/U2/S[2] (net)                            0.00      0.00      3.77 r
ueight_two/U2/S[2] (sixteenFA_0)                              0.00      3.77 r
ueight_two/S[2] (net)                               0.00      0.00      3.77 r
ueight_two/S[2] (eight_two)                                   0.00      3.77 r
S[2] (net)                                          0.00      0.00      3.77 r
uClock_Timer/S[2] (Clock_Timer)                               0.00      3.77 r
uClock_Timer/S[2] (net)                             0.00      0.00      3.77 r
```

```
uClock_Timer/T_S[2] (Clock_Timer)                                 0.00       3.77 r
T_S[2] (net)                                            0.00      0.00       3.77 r
T_S[2] (out)                                                      0.00       3.77 r
data arrival time                                                            3.77

clock CLK (rise edge)                                             4.00       4.00
clock network delay (ideal)                                       0.00       4.00
clock uncertainty                                                -0.20       3.80
output external delay                                             0.00       3.80
data required time                                                           3.80
------------------------------------------------------------------------------------
data required time                                                           3.80
data arrival time                                                           -3.77
------------------------------------------------------------------------------------
slack (MET)                                                                  0.03
```

Appendix 4 – Paper 1 [127]

A (4:2) Adder for Unified GF (p) and GF (2^n) Galois Field Multipliers

Lai Sze Au and Neil Burgess,
Cardiff School of Engineering,
Queens Buildings,
The Parade,
CARDIFF CF24 3TF
U.K.
{auls,burgessn}@cf.ac.uk

Abstract

This paper describes a new redundant binary adder that supports carry-save additions under either of the Galois Fields, GF(p) or GF(2^n), without the need for an extérnal control signal to specify which field is to be used. The proposed adder will find use in unified Galois Field multipliers for cryptographic applications. Its main advantage over previously reported adders is that a control signal which is broadcast to all cells to suppress carries under GF(2^n) is not needed, leading to a substantial gain in implementation efficiency.

1: Introduction

The prime Galois Field GF(p) and the binary extension Galois Field GF(2^n) are the two most important number systems for elliptic curve cryptosystems. The popularity and the need for implementation of dual mode Galois Field Arithmetic operators has increased, due to the interest in inter-operation between different fields. A small number of attempts have been made in recent years to design dual field arithmetic multipliers.

E. Savaş *et al.* [3] proposed a scalable and unified multiplier architecture for finite fields GF(p) and GF (2^n) in 2000, which makes use of Montgomery multiplication to facilitate LSB-first processing.

Figure 1 shows the Processing Unit (PU) of the multiplier. In this design, the operands are required to be transformed into the Montgomery domain.

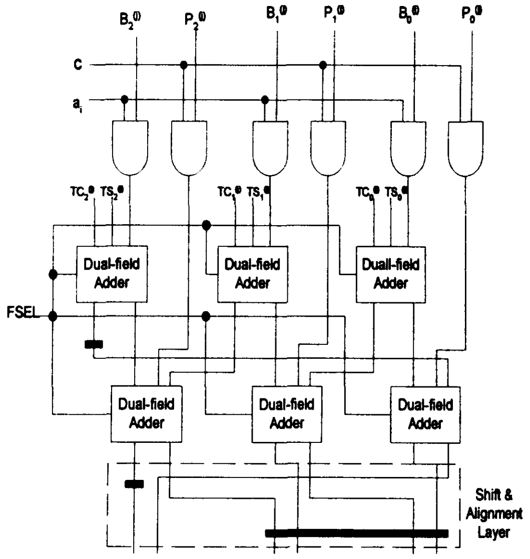


Figure 1 Savaş *et al*'s Processing Unit with $w = 3$

The Dual-field Adder in Figure 1 is a full adder with an extra control signal as shown in Figure 2. In order to perform the dual-field function of the Dual-field Adder, a control signal *FSEL* is needed.

This work was supported by ARM Ltd. URL: www.arm.com
URL: www.arm.com

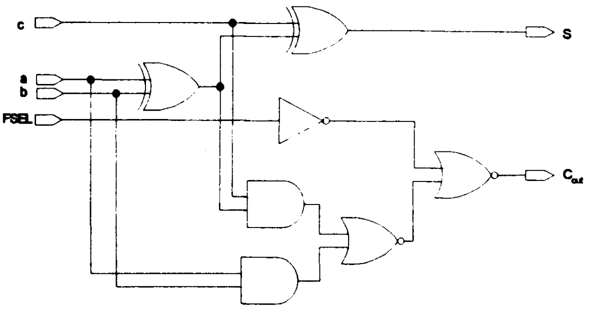


Figure 2 Synthesized circuit of the dual-field adder

When the control signal is 1, the multiplier will perform arithmetic functions in the field of GF(p) and when it is 0, the carry-out will be forced to 0 and the multiplier performs operations in the field of GF(2^n).

Johann Groβshädl [2] proposed a bit-serial unified multiplier architecture for finite field GF(p) and GF(2^n) in 2001 based on an MSB-first iterative algorithm for modulo multiplication.

Figure 3 shows the arithmetic unit that is used for the implementation of the modulo multiplier. The first (n+1)-bit carry-save adder performs the addition of the partial products. The output *Sum* R_s and *Carry* R_c are used to estimate the multiples of modulus to be subtracted in the next step with another (n+1)-bit carry-save adder. Figure 4 is a block diagram of the bit-serial multiplier architecture described in [3]. In order to perform carry-free addition for GF (2^n), all the carry bits of the adder (R_c) are set to 0, which in turn set further control signals. Modulo reduction occurs within the multiplication process by concurrent subtraction of a multiple of the modulus.

Both these proposals require broadcasting a control signal, which is costly and slow, to all the full adder cells so as to suppress all carries in a multiplier. This is especially true when switching between fields, as can occur in a server operating on many different data streams. This paper describes a limited carry propagation adder circuit capable of adding numbers over either a prime Galois field denoted GF(p) or a binary Galois field denoted GF(2^n), but without the need for a control signal to specify which of the two field types is being used.

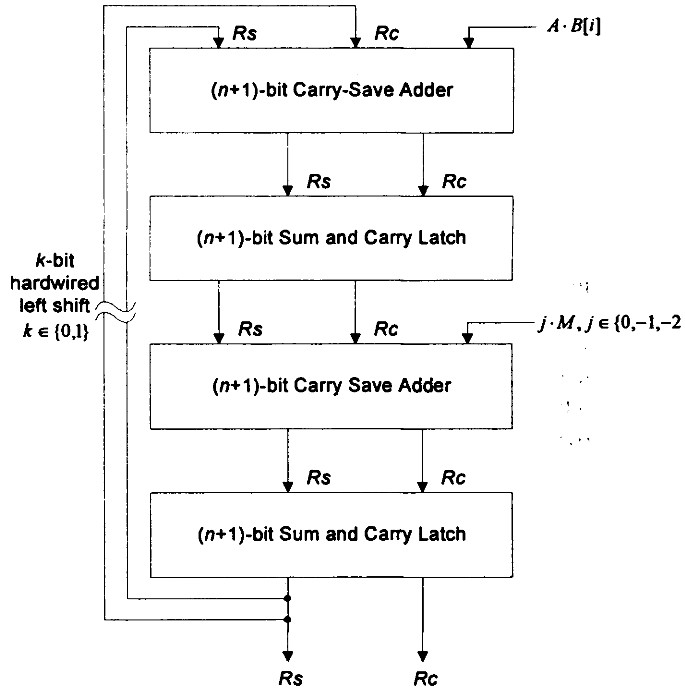

Rs
Rc
A·B[i]
(n+1)-bit Carry-Save Adder
Rs
Rc
(n+1)-bit Sum and Carry Latch
k-bit
hardwired
left shift
k ∈ {0,1}
Rs
Rc
j·M, j ∈ {0,–1,–2
(n+1)-bit Carry Save Adder
Rs
Rc
(n+1)-bit Sum and Carry Latch
Rs
Rc

Figure 3 Arithmetic unit of an n-bit unified multiplier

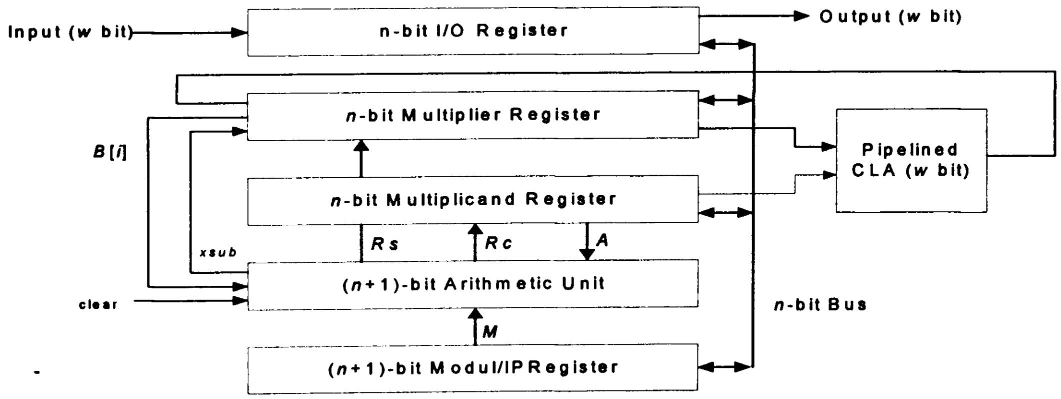

Input (w bit)
n-bit I/O Register
Output (w bit)
n-bit Multiplier Register
B[i]
Pipelined
CLA (w bit)
n-bit Multiplicand Register
Rs
Rc
A
xsub
(n+1)-bit Arithmetic Unit
clear
n-bit Bus
M
(n+1)-bit Modul/IP Register

Figure 4 Block diagram of the bit-serial multiplier architecture

2: Dual Field (4:2) Adder

The main arithmetic operations in prime GF(p) and binary field GF(2^n) are addition, multiplication and inversion. The most important of these operations is the field multiplication operation, formulated as a sequence of additions and subtractions. Although the GF(p) and GF(2^n) fields have different properties, addition operations in the two fields are structurally very similar. The crucial difference between arithmetic over these two types of Galois field is that addition over GF(p) is identical to conventional addition in which carry signals propagate along the length of the sum during the addition, whereas addition over GF(2^n) comprises a bit-wise XOR operation with no carries propagating along the sum. The adder presented here is a so-called "restricted carry" adder which is capable of adding numbers over GF(p) or GF(2^n) in such a way that carries propagate only a limited distance over GF(p) and not at all over GF(2^n).

This section introduces a new dual field adder based on the (4:2) carry-save adder modified so that it is capable of adding specially-encoded operand digits. Hence, no external control signal is needed to suppress carries in GF(2^n) arithmetic. (4:2) adders have been used before in binary multiplier designs as an alternative to carry-save adders because they have more regular multiplier tree layouts. The new adder will permit these same layout advantages to be applied to Galois Field multiplication under either GF(p) or GF(2^n) with some performance advantage over previously reported work.

2.1: Redundant Binary Adder

The Redundant Binary Adder, illustrated in Figure 5, is a binary adder capable of adding two numbers with the digit set $d_i \in \{0, 1, 2\}$ (or equivalently $d_i \in \{-1, 0, 1\}$) such that carries do not traverse the length of the sum [1]. Note that carry signals transform from {0,2} to {0,1} as they shift one place to the left. Each block in the first two rows of Figure 5 can be implemented as a full adder to yield a structure similar to that of Figure 2. The last row of blocks simply concatenate pairs of inputs to provide the output digits.

In the Redundant Binary Adder, digits are implemented using two binary signals (or, in silicon chip terms, wires). If neither signal is 'High' the value '0' is represented; if both signals are 'High' the value '2' is represented; otherwise, if only one signal is 'High' the value '1' is represented (see Table 1). A variety of other coding schemes are possible, but all have the characteristic that only three digits need be represented differently.

Table 1 Redundant Binary Adder Coding

Code	*Digit*
00	0
01	1
10	1
11	2

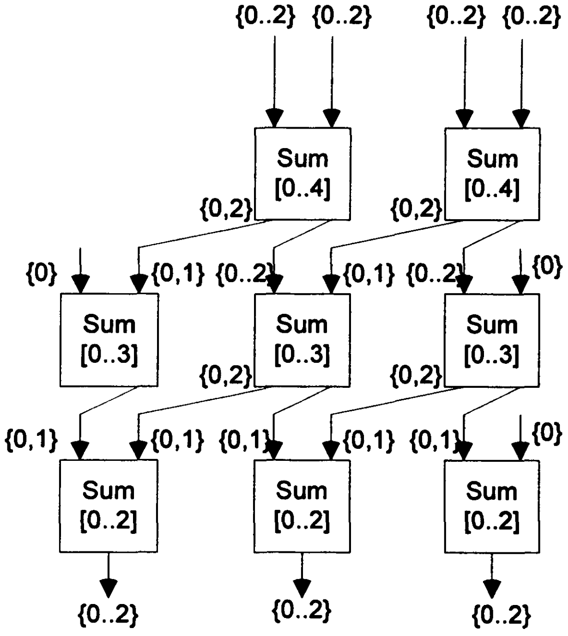


Figure 5 Schematic diagram of a Redundant Binary Adder

2.2: Dual-mode Galois Field Adder

A dual-mode Galois Field adder can be constructed by introducing a fourth digit, denoted 1*, that indicates the digit '1' over GF(2^n). Then, addition over GF(p) is implementable using the digits {0,1,2}, while addition over GF(2^n) is implementable using the digits {0,1*}. Addition over GF(2^n) can be summarised by the expressions: $1^* + 1^* = 0$, and $0 + k = k + 0 = k$. The following digit sets are defined for addition in the two fields:

- for GF(p), 3 values are needed: {0, 1, 2}
- for GF(2^n), only 2 values are needed: {0, 1*}

Therefore, 5 values are apparently needed in total. However, only 4 values are actually needed because the zero elements in both fields are defined identically (see Tables 2 and 3).

Table 2 Table of addition for GF(*p*)

	0	1	2
0	0	1	2
1	1	2	3
2	2	3	4

Table 3 Table of addition of for GF(2^n)

	0	1*
0	0	1*
1*	1*	0

Incorporating the 1* digit into the Redundant Binary Adder is readily accommodated to yield a dual-mode Galois Field adder, as shown in Figure 6. The four symbols {0,1,2,1*} require two wires for their full representation, in common with the redundant binary adder of Figure 1. This enables a unified adder to be constructed, similar in structure to the Redundant Binary Adder (Figure 5). However, the blocks are not now full adders, and so optimum logic circuits for the dual field adder need to be derived.

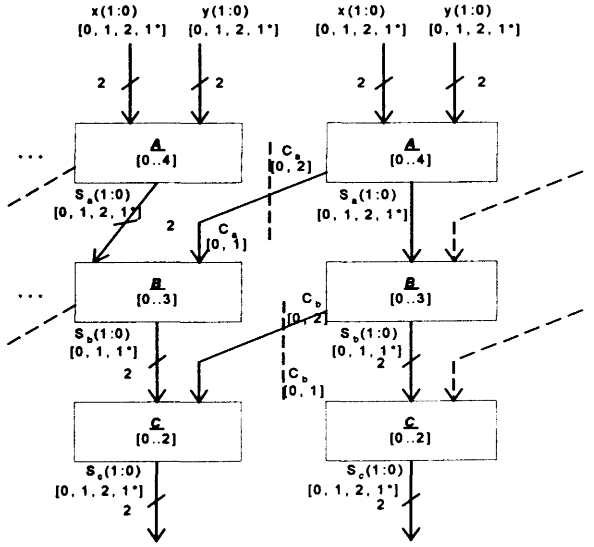


Figure 6 Redundant Dual Field adder

3: Logic Design of Dual Adder

The digits used in the dual adder can be encoded in a variety of different ways. In order to implement the most efficient gate design, i.e. as close to the number of gates required by the original full adder based design as possible, different number encodings are utilised in different cells where needed. Moreover, there are several don't care states which also give some degrees of freedom: for example, the sums 1* + 1 and 1* + 2 cannot occur in any cell. After some experiments, the digit set encoding shown in Table 4 was chosen for the input to cell A.

Table 4 Cell A input coding

Code	*Sum*
00:	0
01:	1*
10:	2
11:	1

The optimal coding for the output sum digit of Cell A was found to be the same as the digit input coding. Also, the output coding of cell C had to be identical to that of cell A, so that the outputs of one dual field adder could be connected directly to the inputs of another in order to realise multiplier designs. The most efficient output sum digit coding for Cell B was found to be different to that of Cells A and C (see Table 5). In this case, the codes for "1" and "1*" were swapped over for the most efficient truth table realisation. The don't care state occurs because the digit "2" is not required by the sum output of Cell B.

Table 5 Cell B output coding

Code	*Sum*
00:	0
01:	1
10:	X
11:	1*

A diagram of the logic circuit of the complete adder is shown in Figure 7.

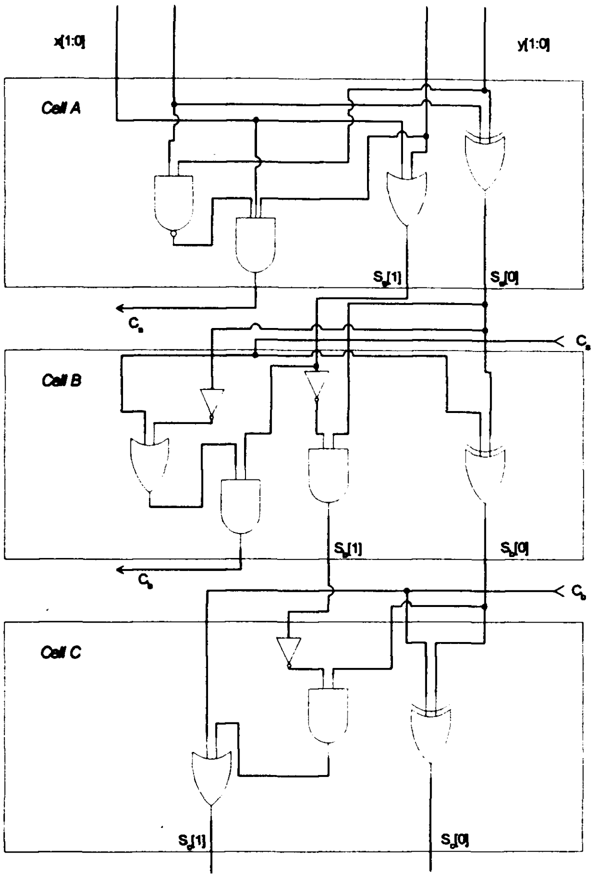


Figure 7 Overall gate implementation of adder

4: Comparisons and Conclusions

This paper has presented a novel (4:2) adder for unified GF(p) and GF(2^n) Galois Field Multiplication. The main difference in implementation between the proposed idea and other previous research is that information regarding the Galois Field under which the addition is to be performed is embedded into the digit coding, obviating the need for a globally-broadcast control signal.

The complete adder of Figure 7 was simulated using NC-Verilog and synthesised using Synopsis, which showed that the critical path (through the three XOR gates) was 1.5 ns using 0.18μm VLSI technology. By comparison, the four-input carry-save adders presented in [3,4] are implemented as pairs of full adders with extra gates on the carry outputs to suppress carries (see Figure 2). Ignoring pipeline stages, these adder cells have a total CMOS logic gate count of 14 (counting XOR gates as two gates) as follows:

- 2 × 2 XOR
- 2 × 1 NOR
- 2 × 1 NOT
- 2 × 1 AOI CMOS complex gate

The proposed adder has a critical path length of only three XOR gates, with a CMOS logic gate count of only 13, made up as follows:

- 3 XOR/XNOR
- 2 NOR
- 1 NAND
- 1 NOT
- 2 OAI CMOS complex gates
- 1 AOI CMOS complex gate

Hence, the new adder is faster and has a simpler field specification mechanism, as well as requiring slightly less logic than previous dual field adders.

Full implementation of a multiplier architecture based on the proposed dual field adder is in progress.

5: References

[1] A. Azivienis, "Signed-Digit Number Representations for fast parallel Arithmetic", *IRE Trans. Elect. Comp.*, EC-10, pp.389-400, Sept.1961

[2] Johann Groβschädl, "A bit-serial unified multiplier architecture for finite fields GF(p) and GF(2^m)", Proc. CHES 2001, Paris, 2001, pp 202-218.

[3] E. Savas, A.F. Tenca, and C.K. Koc, "A scalable and unified multiplier architecture for finite fields GF(p) and GF (2^m)", Proc. CHES 2000, Worcester, MA, August 17-18 2000, pp. 277-295

Appendix 5 – Paper 2 [160]

Unified Radix-4 Multiplier for GF(p) and GF(2^n)

Lai-Sze Au and Neil Burgess
Cardiff School of Engineering,
Queen's Buildings,
The Parade,
CARDIFF CF24 3TF
United Kingdom
{auls, burgessn}@cf.ac.uk

Abstract

This paper describes a scalable unified architecture for Montgomery multiplication over either of the finite fields GF(p) and GF(2^n). This architecture has the advantage of possessing a new redundant binary adder that supports carry-save additions under either of the Galois Fields without the need for an external control signal to specify which field is to be used. Its main advantage over previously reported dual field multiplier is that a control signal which is broadcast to all cells to suppress carries under GF(2^n) is not needed. Consequently, large multipliers can be synthesised whose pipelined speed is independent of the buffering required for the control signal.

1. Introduction

There are two recent trends in multiplier design for cryptographic applications: firstly, the multiplier should be designed as a parallel medium-wordlength architecture so as to increase performance while enhancing resistance to attacks based on differential power analysis [1]; secondly, the multiplier should be capable of operating on either of the popular Galois Field systems [2]. The prime Galois Field GF(p) and the binary extension Galois Field GF(2^n) are the two most important number systems for elliptic curve cryptosystems. The popularity and the need for implementation of dual mode Galois Field Arithmetic operators has increased due to the interest in inter-operation between different fields, and attempts have been made in recent years to design "dual field" Galois field arithmetic multipliers capable of operating under either field [3-5].

E. Savaş *et al.* [3] proposed a scalable and unified multiplier architecture for finite fields GF(p) and GF(2^n) in 2000, which makes use of Montgomery multiplication to facilitate LSB-first processing. Figure 1 shows the Processing Unit (PU) of the multiplier. In this design, the operands are required to be transformed into the Montgomery domain. The Dual-field Adder in Figure 1 is a full adder with an extra control signal as shown in Figure 2. In order to perform the dual-field function of the Dual-field Adder, a control signal *FSEL* is needed. When the control signal is 1, the multiplier will perform arithmetic functions in the field of GF(p)' and when it is 0, the carry-out will be forced to 0 and the multiplier performs operations in the field of GF(2^n). Note that the critical path from any data input (a, b or c) to either output traverses four logic levels, assuming XOR gates have a logic depth of 2.

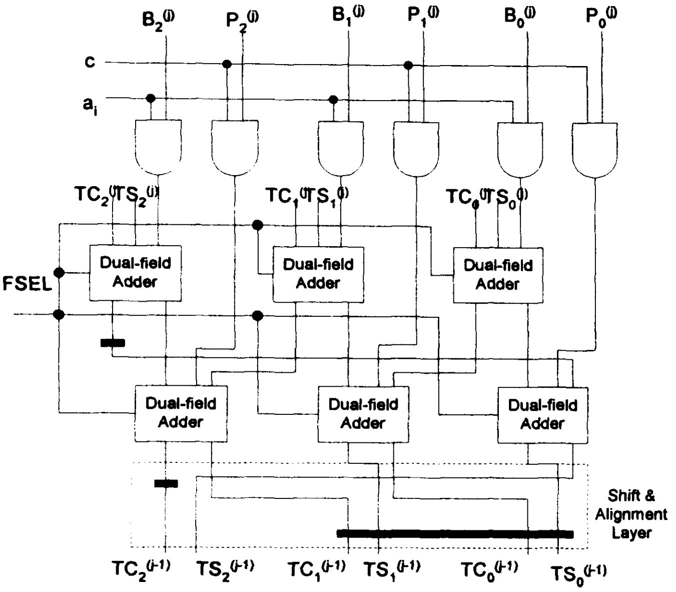


Figure 1. Savaş *et al*'s Processing Unit with_*w* = 3

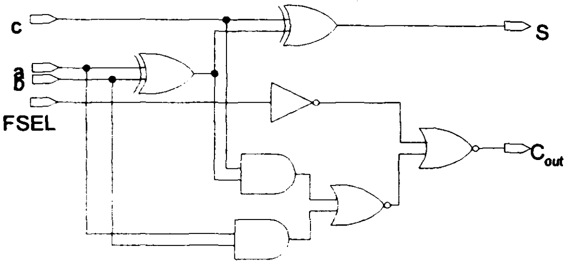


Figure 2. Synthesized circuit of the dual-field adder

Johann Groszshaedl [4] proposed a bit-serial unified multiplier architecture for finite field GF(p) and GF(2^n) in 2001 based on an MSB-first iterative algorithm for modulo multiplication. Figure 3 shows the arithmetic unit that is used for the implementation of the modulo multiplier. The first (n+1)-bit carry-save adder performs the addition of the partial products. The output *Sum* R_s and *Carry* R_c are used to estimate the multiple of the modulus to be subtracted in the next step with another (n+1)-bit carry-save adder.

Figure 4 is a block diagram of the bit-serial multiplier architecture described in [4]. In order to perform carry-free addition for GF(2^n), all the carry bits of the adder (R_c) are set to 0, which in turn set further control signals. Modulo reduction occurs within the multiplication process by concurrent subtraction of a multiple of the modulus.

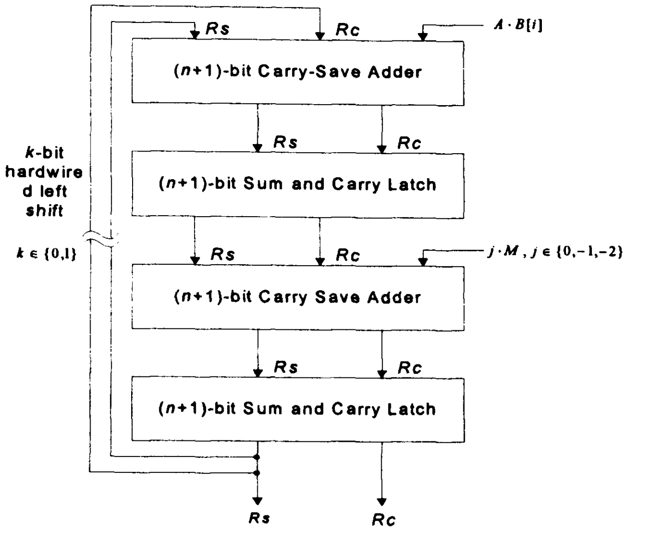

Rs
Rc
$A \cdot B[i]$
(n+1)-bit Carry-Save Adder
Rs
Rc
k-bit hardwired left shift
(n+1)-bit Sum and Carry Latch
$k \in \{0,1\}$
Rs
Rc
$j \cdot M, j \in \{0,-1,-2\}$
(n+1)-bit Carry Save Adder
Rs
Rc
(n+1)-bit Sum and Carry Latch
Rs
Rc

Figure 3. Arithmetic unit of an *n*-bit unified multiplier

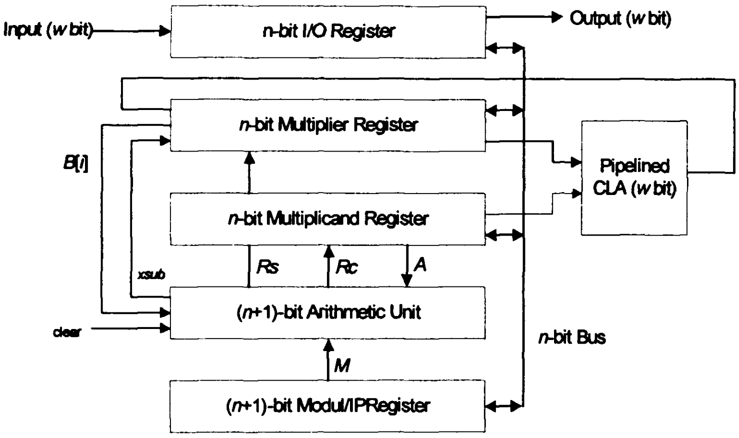

Input (w bit)
n-bit I/O Register
Output (w bit)
n-bit Multiplier Register
B[i]
Pipelined CLA (w bit)
n-bit Multiplicand Register
Rs
Rc
A
xsub
(n+1)-bit Arithmetic Unit
clear
n-bit Bus
M
(n+1)-bit Modul/IPRegister

Figure 4. Block diagram of the bit-serial multiplier architecture

Both these proposals (and others such as [5]) require broadcasting a control signal to all the full adder cells to suppress the output carries from all the full adders in the multiplier. This is costly and slow, especially when switching between fields, as can occur often in a server operating on many different data streams. This paper describes a new multiplier which operates in both GF(p) and GF(2^n). The new multiplier makes use of our previously-presented dual field adder based on a (4:2) carry-save adder cell, modified so that it is capable of adding specially-encoded operand digits [6]. Our unified field multiplier avoids the delay of field-control signal propagation, particularly important as multipliers increase in wordlength for both improved performance and security. The paper is organised as follows: in the next section, we describe the detailed design of the modified (4:2) adder, showing how the particular digit coding schemes employed were selected. Sections 3 and 4 present the full multiplier architecture, together with modules for binary number conversion, partial product generation, and modulo reduction. The paper concludes in Section 5 with a brief discussion of further work.

2. Dual-mode Galois Field Adder Design

(4:2) adders have been used before in binary multiplier designs as an alternative to carry-save adders because they have more regular multiplier tree layouts, requiring less interconnect than other reduction tree topologies [7,8]. The new adder will permit these same layout advantages to be applied to Galois Field multiplication under either GF(p) or GF(2^n). In [6], we described how introducing a fourth digit, denoted 1*, that indicates the digit '1' over GF(2^n) enabled us to take advantage of previously unexploited don't care states in the (4:2) adder cell.

The new (4:2) adder comprises three separate stages, implemented using three different cells: the first stage (cell A) receives two 2-bit operands, $x(1{:}0)$ and $y(1{:}0)$ with the digit set, $d \in \{0,1,2,1^*\}$, and adds them so that the to form a 2-bit sum digit, $s_A \in \{0,1,2,1^*\}$, and a carry bit, $c_A \in \{0,2\}$. The addition is summarised in Table 1, showing that there is considerable flexibility available in the cell's implementation. Specifically, there are four don't care states, and the output digit '2', can be represented by either $s_A = 2$ or $c_A = 2$.

Table 1. Cell A addition

	0	1	2	1*
0	0	1	2	1*
1	1	2	3	X
2	2	3	4	X
1*	1*	X	X	0

Table 2. Cell B addition

	0	1
0	0	1
1	1	2
2	2	3
1*	1*	X

Table 3. Cell C addition

	0	1
0	0	1
1	1	2
X	X	X
1*	1*	X

The second stage (cell B) receives the 2-bit sum digit, s_A, and the shifted carry bit, c_A, from cell A, and adds them to form the 2-bit sum digit, $s_B \in \{0,1,1^*\}$, and a carry bit, $c_B \in \{0,2\}$. The addition is summarised in Table 2, showing that there is less flexibility available in this cell's implementation than in cell A, as there is only one don't care state.

Finally, the third stage (cell C) receives the 2-bit sum digit, s_B, and the shifted carry bit, c_B, output by cell B and adds them to form the 2-bit sum digit, $s_C \in \{0,1,2,1^*\}$. This digit set matches the digit set of the (4:2) adder's inputs, so that the addition is complete [9]. The third stage of the addition is summarised in Table 3, showing that there is more flexibility available in this cell's implementation, due to the increased number of don't care states.

2.1. Cell A digit coding

The digit coding for cell A was chosen as follows: $s_A(0)$ should be a 2-input XOR function, so as to match the delay of a conventional (4:2) adder, and the other two logic functions are required to be as simple as possible. This immediately implies the codes for 1 and 1* should have a Hamming distance of 1 to meet the $s_A(0)$ constraint. After some experiments, we made the assignments (0,1) = 1*, and (1,1) = 1, together with the arbitrary assignment (0,0) = 0, leaving (1,0) = 2. Filling out Table 1 with these digit representations gives the Karnaugh map shown in Table 4, where '-' reflects that the decision about how to represent the output '2' is yet to be made, and 'X' denotes "don't care".

Table 4. Karnaugh Map for Cell A addition

	0 → 00	1* → 01	1 → 11	2 → 10
0 → 00	0, (0,0)	0, (0,1)	0, (1,1)	-, (-,0)
1* → 01	0, (0,1)	0, (0,0)	X, (X,0)	X, (X,1)
1 → 11	0, (1,1)	X, (X,0)	-, (-,0)	1, (1,1)
2 → 10	-, (-,0)	X, (X,1)	1, (1,1)	1, (1,0)

The '-, (-,0)' entries must become either '1, (0,0)' or '0, (1,0)' to represent an output of 2. If they are set consistently to '0, (1,0)', then $s_A(1) = x(1) \vee y(1)$ is obtained by exploiting the don't care states. Finally, by setting all the remaining don't care states for c_A low, we obtain $c_A = x(1) \wedge y(1) \wedge \{\neg x(0) \vee \neg y(0)\}$, implementable as a 2-input NAND driving a 3-input AND, matching the CMOS VLSI delay of the XOR. The final map for cell A is presented in Table 5.

Table 5. Karnaugh Map for Cell A addition

	0 → 00	1* → 01	1 → 11	2 → 10
0 → 00	0, (0,0)	0, (0,1)	0, (1,1)	0, (1,0)
1* → 01	0, (0,1)	0, (0,0)	0, (1,0)	0, (1,1)
1 → 11	0, (1,1)	0, (1,0)	0, (1,0)	1, (1,1)
2 → 10	0, (1,0)	0, (1,1)	1, (1,1)	1, (1,0)

2.2. Cell B digit coding

Using the same coding for cell B as was used in cell A yields $s_B(0) = s_A(0) \oplus c_A$, as required. However, the logic for $s_B(1)$ is not as simple as required with this output encoding. Swapping the output representations for 1 and 1* - that is replacing (0,1) by (1,1) and *vice versa*, did not impact the $s_B(0)$ logic, while simplifying the $s_B(1)$ logic to $s_B(1) = \neg s_A(1) \wedge s_A(0)$. Finally, $c_B = s_A(1) \wedge (\neg s_A(0) \vee c_A)$. The Karnaugh map for cell B is presented in Table 6.

Table 6. Karnaugh Map for Cell

	0	1
0 → 00	0, (0,0)	0, (0,1)
1* → 01	0, (1,1)	0, (1,0)
1 → 11	0, (0,1)	1, (0,0)
2 → 10	1, (0,0)	1, (0,1)

Table 7. Karnaugh Map for Cell

	0	1
0 → 00	(0,0)	(1,1)
1 → 01	(1,1)	(1,0)
1* → 11	(0,1)	(1,0)
X → 10	(0,0)	(1,1)

2.3. Cell C digit coding

This design is straightforward: the output coding must match the input coding of cell A (i.e. 1* → (0,1) and 1 → (1,1)), and the input coding matches the output coding of cell B. By exploiting the don't care states, the logic equations are $s_C(1) = \neg s_B(1) \wedge s_B(0) \vee c_B$, and $s_C(0) = s_B(0) \oplus c_B$. The final Karnaugh map of Cell C is presented in Table 7.

Figure 5 shows the final CMOS gate implementation of the adder, where some further logic optimisation has been made (i) to cover the lack of AND and OR gates in CMOS, and (ii) to take advantage of CMOS complex gates.

3. Multiplier Design

The overall structure of this multiplier is shown in Figure 6. It shows that the multiplier comprises six different modules: (1) Binary to Redundant number encoder; (2) Partial Product Generator; (3) (4:2) adders for partial products summation; (4) Modulus Multiplier Digit Selection; (5) Modulus Multiple Generator; (6) (4:2) adders for modulo reduction. However, only four different modules are required because the two (4:2) adders required are the same as are the partial product and modulus multiple generators are also identical. The (4:2) adder has already been introduced, so this section shall present the design of binary to redundant number encoder and also the design of the partial product generator. The modulo reduction will be presented in the next section of the paper.

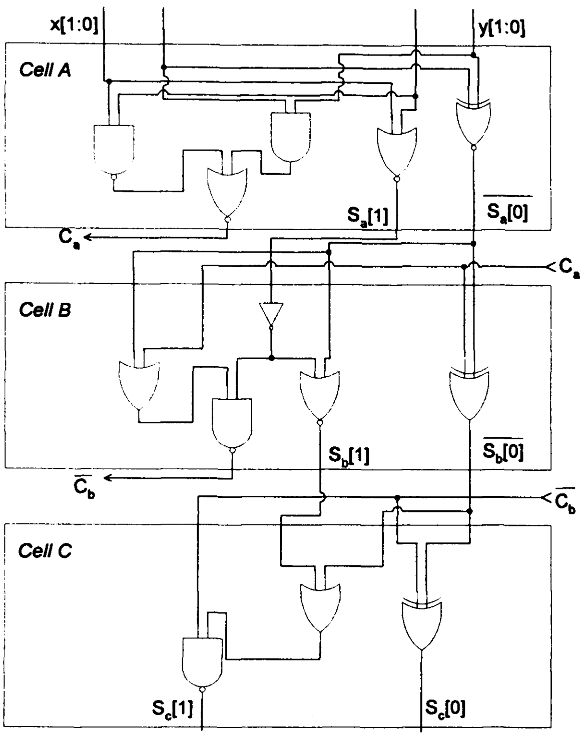


Figure 5. Overall gate implementation of dual field (4:2) adder

The two digits input to the (4:2) adders - namely, the result of the previous iteration, R_n, and the partial product, PP_n, both have the digit set, $d \in \{0, 1, 1^*, 2\}$. The partial product generation is decomposed into two steps: firstly, the selected Galois Field is embedded into the multiplicand word by encoding it using the novel $d = 1^*$ representation; secondly, the radix-4 partial product is derived by using the available redundant $d = 2$ representation. The first of these steps, embedding the Galois Field, is implemented by the simple circuit shown in Figure 7.

Every two bits of the multiplier word, B, are recoded as a radix-4 digit, and the multiplicand, A, then multiplied by the recoded bit to yield the appropriate partial product, as shown in Table 8. Figure 8 shows the logic diagram of the partial product generator, including the Field-embedded binary number encoder, and is seen to be simpler than the standard radix-4 Booth's encoder [10]. In particular, the negative multiple increment bits that occur in Booth's coding are avoided, as these can increase the logic depth of the adder array. Note how the availability of the redundant digit, $d = 2$, at the (4:2) adder input obviates the need for a carry-propagate addition when encoding the radix-4 digit of 3.

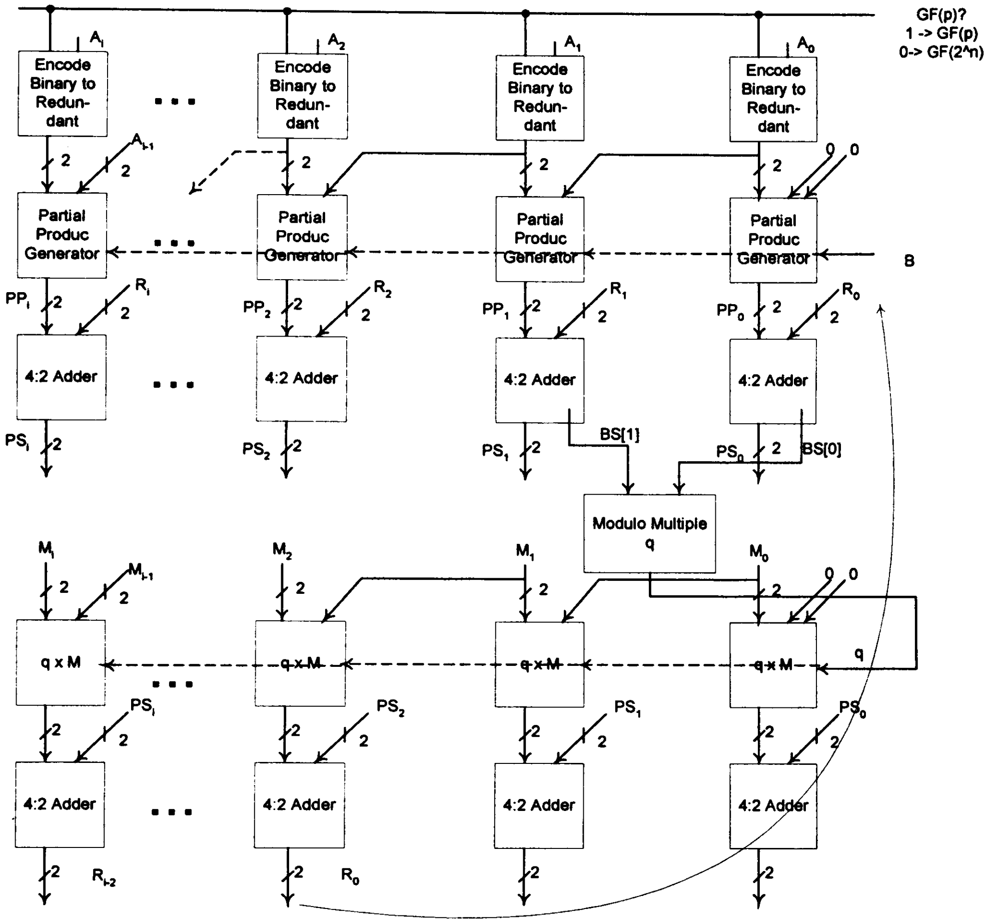


Figure 6. WordxDigit Dual-Field Multiplier Architecture

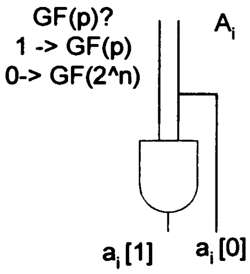


Figure 8. Field-Embedded Binary Number Encoder

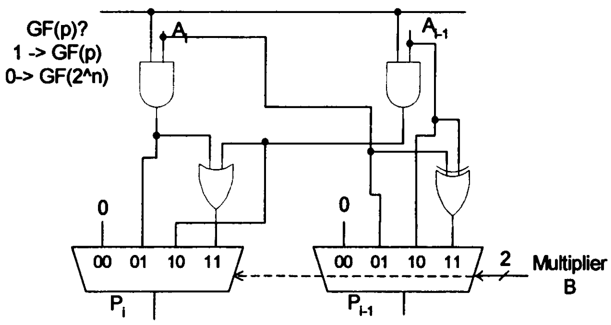


Figure 7. Radix-4 Partial Product Generator

Table 8. Radix-4 Partial Product Generation

(B_i, B_{i-1})	Radix-4 digit	Partial product, $PP_i[1:0]$
00	0	0
01	1	A
10	2	Left shift A 1 bit
11	3	$1A + 2A$

4. Modular Reduction

This design made use of Montgomery's multiplication techniques to perform the modular multiplication. Montgomery's modular multiplication algorithm is described as follows:

Montgomery's Modular Multiplication Algorithm

```
{Pre-condition: $M$ prime to $r$ and $A$ non-redundant}
$S := 0$;
For $i := 0$ to $n-1$ do
Begin
    $q_i := (s_0 + a_i b_o)(-m_o^{-1}) \bmod r$;
    $S := (S + a_i \times B + q_i \times M) div r$;
    {Invariant: $0 \le S < M+B$}
End;
    {Post-condition: $S \times r^n = A \times B + Q \times M$}
```

Figure 6 shows that the modular multiple selection (i.e. determining q_i) causes irregularity in the design and is on the critical path. Therefore, effort is needed to reduce the delay by taking into consideration pre-known factors as early on in the calculation as possible. For example, the modulus M is always an odd number (because $r = 2^n$), so that the last bit of M, M[0], will always be 1. Therefore the information presented in Table 9 regarding the two LSB's of $q_i \cdot M$ is already known before any modulo reductions are performed.

Table 9. Multiple of M

Multiple of M	M[1,0] = 01	M[1,0] = 11
1	01	11
2	10	10
3	11	01

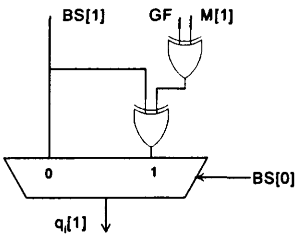


Figure 9. q_i[1] logic

Table 10 shows what value of q_i is required to ensure $R = BS[1, 0] + q_i \cdot M = 00$ as a function of the selected Galois Field, where the two LSB's of the Partial Sum, denoted BS[1:0], are in conventional binary form rather than in redundant form.

Table. 10 Selection of Modulo Multiple, $q_i \cdot M$

	Partial Binary Sum.	q_i[1:0] if M[1,0] = 01	q_i[1:0] if M[1,0] = 11
GF(p), GF = 1	00	00	00
	01	11	01
	10	10	10
	11	01	11
GF(2^n), GF = 0	00	00	00
	01	01	11
	10	10	10
	11	11	01

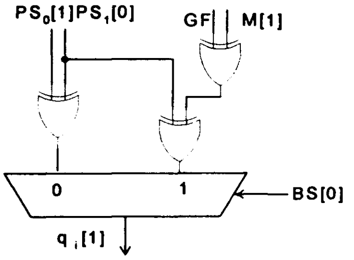
From Table 10, it is easy to see that q_i[0] = BS[0] independently of both the Galois Field and M[1:0]. However, q_i[1] is a function of M[1], BS[1:0], and the Galois Field flag, GF. Figure 9 shows a simple circuit implementing the necessary logic organised as a multiplexer controlled by BS[0].

Montgomery's modular reduction technique is performed on non-redundant binary numbers. Therefore, the redundant representation returned by the (4:2) adders must be converted to binary to obtain the bits *BS*[1:0]. Table 11 presents this conversion process, where PS_i denotes the two bits representing the partial sum at bit position *i* (see Figure 6). Note that PS_1[1] is not included in the Table, because it is weighted +2 and so has no effect on the value of BS[1].

Table 10.1. Binary Conversion

$PS_1[0]$	$PS_0[1]$	$PS_0[0]$	digit[1]	digit[0]	BS [1]	BS [0]
0	0	0	0	0	0	0
0	0	1	0	1*	0	1
0	1	0	0	2	1	0
0	1	1	0	1	0	1
1	0	0	1*	0	1	0
1	0	1	1*	1*	1	1
1	1	0	1*	2	0(x)	0(x)
1	1	1	1*	1	1(x)	1(x)
0	0	0	2	0	0	0
0	0	1	2	1*	0(x)	1(x)
0	1	0	2	2	1	0
0	1	1	2	1	0	1
1	0	0	1	0	1	0
1	0	1	1	1*	1(x)	1(x)
1	1	0	1	2	0	0
1	1	1	1	1	1	1

The Table shows that *BS*[0] = PS_0[0]. In fact, since C_a and C_b to the (4:2) adder are both 0, BS[0] = PP_0[0] ⊕ R_0[0], and is available much earlier than BS[1]. The logic for BS[1] is presented in Figure 10 as a multiplexer controlled by BS[0], in common with Figure 9. Merging Figures 9 and 10 yields the simplified circuit for q_i[1] shown in Figure 11.



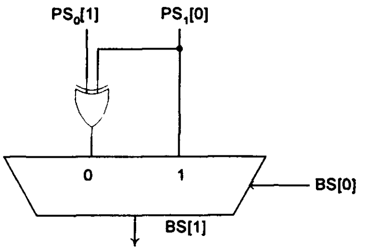


Figure 10. Logic for BS[1]

Figure 11. Simplified logic for q_i[1]

Once q_i has been determined, the modulo *M* is multiplied by q_i using the modulo multiple generator shown in Figure 12, which has the same logic design as in the partial product generator presented earlier. Finally, the multiple, $q_i \cdot M$, is then added to partial sum (*PS*) using the same modified (4:2) adder as shown in Figure 5. Note that in Figure 6, the least significant four bits (2 binary bits) are discarded as they are now zero and what was R_2 is now fed back to the partial product adder as R_0.

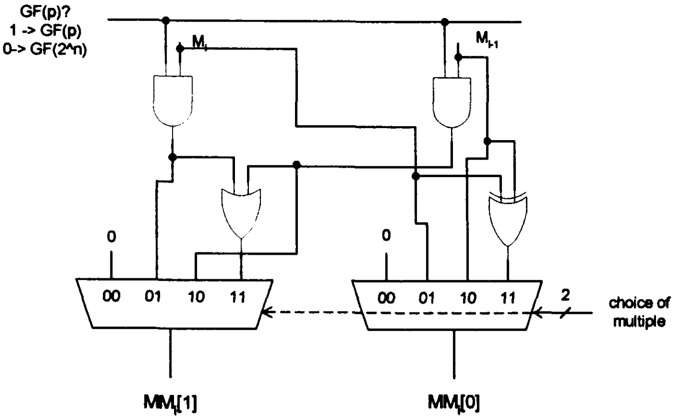


Figure 12. Modulo multiple generator

5. Comparisons

When compared with E. Sava° *et al.*'s design [5], previously shown in Figure 1, the unified multiplier presented here has the advantage that the Galois Field selection line does not cause extra delays due to potentially large fanouts. In Figure 1, the FSEL line has to drive $2w$ NOT gates in the dual field adders, where w is the word-length of the adder.

The delay of the FSEL line driving $2w$ inverters can be estimated by using Logical Effort [11] as being roughly $\log_4 2w$ FO4 delays, where FO4 denotes "fanout of 4 inverters". The pipeline delay comprises this buffer delay and the adder delay, assuming that the partial product is gener-ated in a prior pipeline stage.

In Figure 2, there are two critical paths through the adder: one starts with inputs a and b and traverses an XOR gate, a (2,2) AND-OR-invert (AOI) gate, and a NOR gate; the other starts with the FSEL line and comprises the FSEL buffer, an inverter, and the same NOR gate as the other path. The FSEL delay dominates the pipeline stage when the buffer delay becomes larger than the delay of the XOR combined with the AOI gate. From Logical Effort, the delay of the buffer was found to be $\log_4 w$; the delay of an XOR gate (assuming its implementation by a CMOS (2,2) AOI gate) and a second (2,2) AOI gate is given by $2((2(1 + 4)/5 = 2.4$ FO4, using the equation $d = gh + p$ and inserting the relevant values. Therefore when $\log_4 2w > 2.4$, or $w = 14$, the FSEL buffer delay starts to dominate the critical path and affects the maximum clock rate achievable. More-over, if only one bit is processed per pipeline stage, then this design could be vulnerable to Power Analysis cryptographic attacks as the word-length is small [1]. However, increasing the number of bits per stage increases the fanout on the FSEL line, further degrading performance.

The word-length in the proposed design can be increased per pipeline stage as much as needed without causing extra delay so that this design is truly scalable. Moreover, this design could proc-ess more than one digit per pipeline stage without any extra delay due to field selection, although there would be additional delay due to the extra adders in each pipeline stage. However, the Mul-tiplicand A and the Modulus M need to be converted into the novel redundant number coding, but this can be done in parallel with the Multiplier B being fed to the row of partial product genera-tors, thus avoiding any delay due solely to field selection.

In terms of area, E. Sava° *et al.*'s design requires two extra gates per full adder for field selec-tion. The proposed design requires w extra AND gates in the partial product and modulus multi-ple generators to embed the field information into the digits, while the unified field (4:2) adders have the same number of logic gates as two full adders [6]. Thus, the area of the proposed design matches that of previous designs.

6. Conclusion and future work

This paper has presented a Montgomery modulo multiplier using a novel (4:2) adder for uni-fied GF(p) and GF(2_n) Galois Field Multiplication. The main difference in implementation be-tween the proposed idea and other previous research is that information regarding the Galois Field under which the addition is to be performed is embedded into the digit coding, obviating the need for a globally-broadcast control signal. Also, both the partial product generation and modulo reduction are performed using radix-4 algorithms to accelerate the processing time.

The critical path of this architecture, from the input to the first row of (4:2) adders to the out-put of the second row of (4:2) adders consists of 7 XOR gates and 2 multiplexers, of which 6 XOR gates are in the two (4:2) adders. This logic depth compares favourably with the radix-2 unified adders presented in Section 1 of this paper, which have a critical path of 8 logic levels excluding the logic needed to derive q_i. We are currently implementing the proposed design in CMOS VLSI.

Acknowledgement

This work was supported by ARM Ltd., Fulbourn Road, Cambridge, U.K.

References

[1] C. D. Walter, "Techniques for the Hardware Implementation of Modular Multiplication", Proc. 2nd IMACS Int. Conf. on Circuits, Systems & Computers, Athens, October 1998, vol. 2, pp 945-949.

[2] J. Goodman and A.P. Chandrakasan, "An Energy-efficient Reconfigurable Public-key Cryptography Processor", *IEEE J. Solid-State Circuits*, vol. 36, pp. 1808-1820 (November 2001)

[3] E. Savas, A.F. Tenca, and C.K. Koc, "A scalable and unified multiplier architecture for finite fields GF(p) and GF(2^m)", Proc. CHES 2000, Worcester, MA, August 2000, pp. 277-295

[4] Johann Groβschädl, "A bit-serial unified multiplier architecture for finite fields GF(p) and GF(2^m)", Proc. CHES 2001, Paris, August 2001, pp 202-218.

[5] J. Wolkerstorfer, "Dual-Field Arithmetic Unit for GF(p) and GF(2^m)", Proc. CHES 2002, San Francisco, August 2002

[6] L.S. Au, N. Burgess "A (4:2) Adder for Unified GF(p) and GF(2^n) Galois Field Multipliers", Proc. 36th Asilomar Conference on Signals, Systems and Computers, November 2002

[7] N. Takagi, H. Yasura and S. Yajima, "High-speed VLSI multiplication algorithm with a redundant binary addition tree", *IEEE Transactions on Computers*, vol. 34, pp. 789-796 (August 1985)

[8] M.R. Santoro and M.A. Horowitz, "SPIM: a pipelined 64×64-bit iterative multiplier", *IEEE Journal of Solid-State Circuits*, Vol. 24, pp. 487-493 (April 1989)

[9] A. Azivienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic", *IRE Trans. Elect. Comp.*, vol. EC-10, pp.389-400, (September 1961)

[10] J. Groszschaedl, "A unified radix-4 partial product generator for integers and binary polynomials" Proc. IEEE ISCAS, Scottsdale, AZ, pp. 567 - 570 vol.3 (May 2002)

[11] I. Sutherland, B. Sproull, C. Harris, "Logical Effort – Designing Fast CMOS Circuits", Morgan Kaufmann Publishers, (1999)

References

[1] Walter C. D, "Longer Keys may facilitate Side Channel Attacks", Proceedings of the 10th Annual Workshop on Selected Areas in Cryptography - SAC 2003, p. 14-15, 2003

[2] Smith J.L., "The Design of Lucifer, A Cryptographic Device for Data Communications", IBM Research Report RC3326, 1971

[3] National Bureau of Standards, "Data Encryption Standard", Federal Information Processing Standards Publication FIBS PUB 46, 1977

[4] ANSI X3.92, "American National Standard for Data Encryption Algorithm (DEA)", American National Standards Institute, 1981

[5] http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[6] http://www.semiconductors.philips.com/markets/identification/datasheets/index.html

[7] http://www.fecinc.co.jp/pdf/IC_CHIP.PDF

[8] Preissig R. S., "Data Encryption Standard (DES) Implementation on the TMS320C6000", Texas Instruments Application report SPRA702, 2000 http://www.ee.ic.ac.uk/pcheung/teaching/ee3_Study_Project/DES%20Implementation(702).pdf

[9] Lai X. and Massey J., "A Proposal for a NEW Block Encryption Standard", proceedings of Advances in Cryptology – EUROCRYPT'90, Springer-Verlag, p. 389-404, 1991

[10] Lai X., Massey J. and Murphy S., "Markov Ciphers and Differential Cryptanalysis", Proceedings of Advances in Cryptology – EUROCRYPT'91, Springer-Verlag, p. 17-38, 1991

[11] http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[12] Daemen J. and Rijmen V., "The Rijndael Block Cipher- AES Proposal", First AES Candidate Conference (AES1), 1998

[13] Diffie W. and Hellman M., "New directions in cryptography", IEEE Transactions on Information theory, Vol. 22, p. 644-654, 1976

[14] Maurer U., "Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms", Advances in Cryptology - Crypto '94, Springer-Verlag, p. 271-281, 1994

[15] Diffie W., van Oorschot P.C., and Wiener M.J. "Authentication and authenticated key exchanges", Designs, Codes and Cryptography, vol 2, p. 107-125, 1992

http://www3.sympatico.ca/wienerfamily/Michael/MichaelPapers/STS.pdf

[16] http://www.itl.nist.gov/fipspubs/fip180-1.htm

[17] ElGamal T., "A public key cryptosystem and a signature scheme based on discrete logarithms", Proceedings of Crypto '84, LNCS, vol. 196, p. 10-18, 1984

[18] http://www.itl.nist.gov/fipspubs/fip186.htm

[19]: Rivest R., Shamir A. and Adleman L., "A method for obtaining digital signatures and public-key cryptosystems", Communications of the ACM, vol. 21, p. 120-126, 1978

[20] Rivest R., Shamir A. and Adelman L., "On Digital Signatures and Public Key Cryptosystems", MIT Laboratory for Computer Science Technical Memorandum 82, 1977

[21] http://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1014_Menezes.sigs.pdf

[22] http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf

[23] ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.doc

[24] American National Standards Institute, "ANSI X9.31-1998: Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry (rDSA)", 1998

[25] Rivest R. L. and Kaliski B., " RSA Problem", MIT Laboratory for Computer Science & RSA Laboratories, 2003
http://theory.lcs.mit.edu/~rivest/RivestKaliski-RSAProblem.pdf

[26] Boneh D. and Venkatesan R., "Breaking RSA may not be equivalent to factoring", Proceedings Of Eurocrypt'98, LNCS, Springer-Verlag, vol. 1233, p. 59-71, 1998
http://theory.stanford.edu/~dabo/papers/no_rsa_red.pdf

[27] Boneh D., "Twenty years of attacks on the RSA cryptosystem", Notices of the AMS, vol. 46, p. 203-213, 1999
http://www.ams.org/notices/199902/boneh.pdf

[28] Davida G., "Chosen signature cryptanalysis of the RSA (MIT) public key cryptosystem", Techical Report: TR-CS-82-2, Dept. of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, Wisconsin, 1982

[29] Denning D. E., "Digital signatures with RSA and other public-key cryptosystems", Comm. ACM 27, vol. 27, p. 388-392, 1984

[30] Desmedt Y. and Odlyzko A. M., "A chosen text attack on the RSA cryptosystem and some discrete logarithm schemes", Proceedings CRYPTO'85, Springer-Verlag, LNCS, vol 218, p. 516-522, 1986
http://www.dtc.umn.edu/~odlyzko/doc/arch/rsa.attack.pdf

[31] Bellare M. and Rogaway P., "Optimal asymmetric encryption-how to encrypt with RSA", Proceedings Eurocrypt'94, Springer-Verlag, LNCN, vol. 218, p. 92-111, 1994

[32] Okamoto T. and Pointcheval D., "REACT: Rapid enhanced-security asymmetric cryptosystem transform", Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographers' Track RSA Conference, Springer-Verlag, p. 159-175, 2001

[33] Shoup V., "A Proposal for an ISO Standard for Public Key Encryption (version 2.1)", 2001
http://shoup.net/papers/

[34] Hastad J., "Solving simultaneous modular equations of low degrees", SIAM Journal on Computing: Special issue on cryptography, vol. 17, p. 336-341, 1988

[35] Boneh D. and Durfee G., "Cryptanalysis of RSA with private key *d* less than $N^{0.292}$", IEEE Transactions on Information Theory, vol. 46, p. 1339-1349, 2000

[36] Wiener M., "Cryptanalysis of short RSA secret exponents", IEEE Transactions on Information Theory, Vol. 36, p. 553-558, 1990

[37] Zimmermann P. , "The Official PGP User's Guide", The MIT Press, 1995

[38] NIST, "Key Management Guideline - Workshop Document" Draft, 2001.
http://csrc.nist.gov/encryption/kms/key-management-guideline-(workshop).pdf

[39] American National Standards Institute, "ANSI X9.52-1998: Triple Data Encryption Algorithm Modes of Operation", 1998

[40] NIST, "Secure Hash Standard", 2002.
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[41] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using discrete Logarithm Cryptography", 2001

[42] NIST, "DRAFT Special Publication 800-56, Recommendation on Key Establishment Schemes" Draft, 2003

[43] RSA Laboratories, "PKCS #1 v2.0: RSA Cryptography Standard", 1998.

[44] American National Standards Institute, "Key Management Using Reversible Public Key Cryptography for the Financial Services Industry", Work in Progress

[45] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)", 1999

[46] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry, Key Agreement and Key Transport Using Elliptic Curve Cryptography", 1999

[47] NIST, "DRAFT Special Publication 800-38b, Recommendation for Block Cipher Modes of Operation: The RMAC Authentication Mode" Draft, 2002

[48] NIST, "The Keyed-Hash Message Authentication Code (HMAC)", 2002

[49] Bernstein D.J., "Circuits for Integer Factorization: A Proposal", Manuscript, 2001.
http://cr.yp.to/papers.html#nfscircuit

[50] Shamir A. and Tromer E., "Factoring Large Numbers with the TWIRL Device", Proceedings of Crypto 2003, Springer-Verlag, LNCS, vol 2729, p. 1-26, 2003
http://www.wisdom.weizmann.ac.il/~tromer/papers/twirl.pdf)

[51] http://www.rsasecurity.com/rsalabs/technotes/twirl.html

[52] NESSIE Consortium, "Portfolio of recommended cryptographic primitives", 2003.
https://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/decision-final.pdf

[53] Miller V., "Uses of elliptic curves in cryptography" Advances in Cryptology, CRYPTO '85, Springer-Verlag, LNCS, vol 218, p. 417-426, 1986

[54] Koblitz N., "Elliptic Curve Cryptosystems", Mathematics of Computation, vol. 48, p. 203-209, 1987

[55] Schroeppel R., Orman H. and O'Malley S., "Fast Key Exchange with Elliptic Curve Systems", Advances in Cryptography, Crypto '95, Springer-Verlag, LNCS, vol. 963, p. 43-56, 1995
http://www.zone-h.org/files/33/TR95-03.pdf

[56] Robshaw M.J.B. and Yin Y. L., "Elliptic Curve Cryptosystems", An RSA Laboratories Technical Notes, revised 1997.
http://www.rsasecurity.com/rsalabs/technotes/elliptic_curve.html

[57] Choie Y., Jeong E., Lee E., "Supersingular Hyperelliptic curve of Genus 2 over Finite Fields", Cryptology ePrint Achive: Report, 2002

http://math.postech.ac.kr/~ancy/pub/journal/ss_g2_update.pdf

[58] Wiener M.J. and Zuccherato R.J., "Faster Attacks on Elliptic Curve Cryptosystems", Springer-Verlag, LNCS, vol. 1556, p. 190, 1999. http://www3.sympatico.ca/wienerfamily/Michael/MichaelPapers/ECattack.pdf

[59] Montgomery P. L., "Speeding the Pollard and Elliptic Curve Methods of Factorization", Mathematics of Computation, vol. 48, p. 243-264, 1987

[60] Okeya K., Kurumatani,H. and Sakurai,K., "Elliptic Curves with the Montgomery-Form and Their Cryptographic Applications", Public Key Cryptography (PKC 2000), LNCS, vol. 1751, p. 238-257, 2000

[61] Kocher C, "Cryptanalysis of Diffie-Hellman, RSA, DSS, and Other Systems Using Timing Attacks", Proceedings of Advances in cryptology, CRYPTO '95: 15th Annual International Cryptology Conferences, Santa Barbara, California, USA, Sringer-Verlag, p.171-183, 1995 http://www.cryptography.com

[62] Kocher C, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", Advances in Cryptology – CRYPTO '96, LNCS, vol. 1109, p. 104-113, 1996

[63] HITACHI, "Key Agreement scheme OK-ECDH", 2001. http://www.sdl.hitachi.co.jp/crypto/ok-ecdh/index.html

[64] Okeya K. and Sakurai K, "Efficient Elliptic Curve Cryptosystems from a Scalar Multiplication Algorithm with Recovery of the *y*- Coordinate on a Montgomery-Form Elliptic Curve", Cryptographic Hardware and Embedded System (CHES 2001), LNCS, vol. 2162, p. 126-141, 2001

[65] Silverman J.H., "The Arithmetic of Elliptic Curves", Grad. Texts in Mathematics, Springer-Verlag, vol 106, 1986

[66] Menezes A. J., "Elliptic Curve Public Key Cryptosystems", Kluwer Academic Publishers, 1993

[67] Hankerson D., "Performance comparisons of elliptic curve systems in software", 5th workshop on Elliptic Curve Cryptography (ECC 2001), University of Waterloo, Canada, p. 2001. http://www.cacr.math.uwaterloo.ca/conferences/2001/ecc/hankerson.pdf

[68] Leung I., "A Microcoded Elliptic Curve Cryptographic Processor", Mphil thesis, The Chinese University of Hong Kong, 2001 http://www.cse.cuhk.edu.hk/~khleung/thesis/thesis.html

[69] Chudnovsky D.V., "Sequences of numbers generated by addition in formal groups and new primality and factorisation tests", Advances in Applied Math, p. 385-434, 1986

[70] Cohen H., Miyaji A. and Takatoshi O., "Efficient Elliptic Curve Exponentiation Using Mixed Coordinates", Proceedings Of the International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology, LNCS, vol. 1514, p. 51-65, 1998

[71] Paar C., "Implementation options for finite field arithmetic for elliptic curve cryptosystems", invited presentation at the 3rd workshop on elliptic Curve Cryptography (ECC'99), University of Waterloo, Waterloo, Ontario, Canada, 1999
http://www.crypto.ruhr-uni-bochum.de/Publikationen/texte/paar_ecc99.pdf

[72] De Win E., Bosselaers A., Vendenbergh S., De Gersem P. and Vandewalle J., "A fast software implementation for arithmetic operations in GF(2^n)", Proceedings of Asiacrypt'96, LNCS, vol. 1163, p. 65-76, 1996

[73] Hankerson D., Hernandez J.L. and Menesez A., "Software Implementation of Elliptic Curve Cryptography over Binary Fields", Proceedings of CHES' 2000, Spring-Verlag, LNCS, vol. 1965, p. 1-24, 2000.

[74] López J. and Dahab R., "High-Speed software multiplication in F($2m$)", Proceedings of the First International conference on Progress in cryptology, Springer-Verlag, LNCS, vol. 1977, p. 203-212, 2000.
http://www.dcc.unicamp.br/ic-main/publications-e.html

[75] Ash D., Blake I. and Vanstone S.A., "Low Complexity Normal Basis", Discrete Applied Mathematics, Vol. 25, p. 191-210, 1989

[76] Mullin R. C., Onyszchuk I. M., Vanstone S. A. and Wilson R., "Optimal Normal Basis in GF(p^m)", Discrete Applied Mathematics, Vol. 22, p. 149-161 1988

[77] Massey J.L. and Omura J.K., " Computational method and apparatus for finite field arithmetic", US patent # 4587627, 1986

[78] Mullin R.C., "Multiple Bit Multiplier", US Patent # 5787028, 1998

[79] Mullin R.C, Onyszchuk I. M. and Vanstone S. A., "Computational Method and Apparatus for Finite Field Multiplication", US Patent # 4745568, 1988

[80] http://www.certicom.com

[81] Johnson D. and Menezes A., "The Elliptic Curve Digital Signature Algorithm (ECDSA)", Technical Report CORR 99-34, Dept. of C&O, University of Waterloo, Canada, 2000
http://www.cacr.math.uwaterloo.ca

[82] Hankerson D. and Menezes A., "Elliptic Curve Discrete Logarithm Problem", Technical notes, Technische universiteit eindhoven, 2003

http://www.win.tue.nl/~henkvt/ecdlp.pdf

[83] Pohlig S. and Hellman M., "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance", IEEE Transactions on Information Theory, vol. 24, p. 106-110, 1978

[84] Pollard J., "Monte Carlo methods for index computation mod *p*", Mathematics of Computation, vol. 32, p. 918-924, 1978

[85] Teske E., "Speeding up Pollard's rho method for computing discrete logarithms", Proceedings of the Third International Symposium on Algorithmic Number Theory, Springer-Verlag, LNCS, vol. 1423, p. 541-554, 1998

[86] Van Oorschot P. and Wiener M. J., "Parallel collision search with cryptanalytic applications", Journal of Cryptology, vol. 12, p. 1-28, 1999

[87] Silverman R. and Stapleton J., Contribution to ANSI X9F1 working group, 1997

[88] Menezes A., Okamoto T., and Vanstone S., "Reducing elliptic curve logarithms to logarithms in a finite field", IEEE Transactions on Information Theory, vol. 39, p. 1639-1646, 1993

[89] Frey G. and Rück II., "A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves", Mathematics of Computation vol. 62, p. 865-871, 1991

[90] Frey G., "How to disguise an elliptic curve (Weil descent)", talk at ECC'98, University of Waterloo, Canada, 1998
http://www.cacr.math.uwaterloo.ca

[91] Frey G., "Applications of arithmetical geometry to cryptographic instructions", Proceedings of the Fifth International Conference on Finite Fields and Applications, Springer-Verlag, p. 128-161, 2001

[92] Gaudry P., Hess F. and Smart N. P., "Constructive and Destructive Facets of Weil Descent on Elliptic Curves", Technical Report CSTR-00-016, Department of Computer Science, University of Bristol, 2000

[93] Semaev I., "Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic *p*", Mathematics of computation, vol. 67, p. 353-356, 1998

[94] Smart N., "The discrete logarithm problem on elliptic curves of trace one", Journal of Cryptology, vol. 12, p. 193-196, 1999

[95] Satoh T. and Araki K., "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves", Commentarii Mathematici

Universitatis Sancti Pauli, vol. 47, p. 81-92, 1998

[96] Adlman L., DeMarrais J. and Huang M., "A subexponential algorithm for discrete logarithms over the rational subgroup of the Jacobians of large genus hyperelliptic curves over finite fields", Algorithmic Number Theory, Springer-Verlag, LNCS, vol. 877, p. 28-40,1994

[97] Enge A. "Computing Discrete Logarithms in High-Genus Hyperelliptic Jacobians in Provably Subexponential Time", Mathematics of Computation Col. 71, p. 729-742, 2002

[98] Silverman J. and Suzuki j., "Elliptic curve discrete logarithms and the index calculus", Advances in Cryptology – Asiacrypt '98, Springer-Verlag, LNCS, vol. 1514, p. 110-125, 1999

[99] Kocher C., "Timing Attacks on Implementations of Diffie-Helman, RSA, DSS, and Other Systems", Advances in Cryptology – CRYPTO'96, LNCS, vol. 1109, p. 104-113, 1996

[100] Kocher P., Jaffe J. and Jun B., " Differential Power Analysis", CRYPTO'99, Springer-Verlag, LNCS, vol. 1666, p. 388-397, 1999

[101] Fischer W., Giraud C., Knudsen E. W. and Seifert J-P, "Parallel scalar multiplication on general elliptic curves over F_p hedged against non-differential side-channel attacks", Cryptology ePrint Archive, Report 2002/007, IACR, 2002.
http://eprint.iacr.org

[102] Coron J-S, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems", In workshop on cryptographic hardware and embedded systems – CHES'99, Springer-Verlag, LNCS, vol. 1717, p. 292-302, 1999
http://www.gemplus.com/smart/r_d/publications/pdf/Cor99dpa.pdf

[103] Kömmerling O. and Kuhn M., "Design Principles for Tamper-Resistant Smartcard Processors", Processdings of USENIX Workshop on Smartcard Technology, Chicago, p. 9-20, 1999

[104] Goubin L., "A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems", In Workshop on Practice and Theory in Public Key Cryptosystems (PKC), Spring-Verlag, LNCS, vol. 2567, p. 199-210, 2003

[105] Izu T. and Takahi T., "Exceptional Procedure Attack on Elliptic Curve Cryptosystems", In Workshop on Practice and Theory in Public Key Cryptosystem (PKC), Springer-Verlag, LNCS, vol. 2567, p. 224-239, 2003

[106] Biehl I., Meyer B. and Müller V., "Differential Fault Attacks on Elliptic Curve Cryptosystems", Proceedings of Advances in Cryptology (CRYPTO), Springer-Verlag, LNCS, vol. 1880, p. 131-146, 2000

[107] Clavier C. and Joye M., "Universal Exponentiation Algorithm", Workshop on Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag LNCS 2162, p. 300-308, 2001

[108] Brier É. And Joye M., "Weierstrass Elltipic Curves and Side-Channel Attacks", Workshop on Practice and Theory in Public Key Cryptosystem (PKC), Springer-Verlag, LNCS, vol. 2274, p. 335-345, 2002

[109] Joye M. and Quisquater J-J., "Hessian Elliptic Curves and Side-Channel Attacks", Proceedings of Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag, LNCS, vol. 2162, p. 402-410, 2001

[110] Liardet P-Y and Smart N.P. "Preventing SPA/ DPS in ECC Systems Using the Jacobi Form", Proceedings of Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag, LNCS, vol. 2162, p. 391-401, 2001

[111] Joye M. and Yen S M. "The Montgomery Powering Ladder", Proceedings of Hardware and Embedded Systems – CHES 2002, Springer-Verlag, LNCS, vol. 2523, p. 291-302, 2003

[112] Ha J.C. and Moon S.J., "Randomised Signed-Scalar Multiplication of ECC to Resist Power attacks", workshop on cryptographic hardware and embedded systems (CHES), Spring-Verlag, LNCS, vol. 2523, p. 551-563, 2002

[113] Oswald E. and Aigner M. "Randomised Addition-Subtraction Chain as a Countermeasure Against Power Attacks", workshop on cryptographic hardware and embedded systems (CHES), Springer-Verlag, LNCS, vol. 2162, p. 39-50, 2001

[114] Joye M. and Tymen C. "Protections Against Differential Analysis for Elliptic Curve Cryptography – An Algebraic Approach", workshop on cryptographic hardware and embedded systems (CHES), Springer-Verlag, LNCS, vol. 2162, p. 377-290, 2001

[115] Guajardo J., Wollinger T. and Parr C. "Area efficient GF(p) Architectures for GF(p^m) Multipliers", Proceedings of 45th IEEE International Midwest Symposium on Circuits and Systems – MWSCAS 2002, Tulsa, Oklahoma, vol. 2, p. 37-40, 2002

[116] Gutub A. A-A., Tenca A.F. and Koc C.K. "Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation", IEEE Computer Society Annual Symposium on VLSI, Pittsburgh, Pennsylvania, p. 53-58, 2002

[117] Wu H., "Montgomery Multiplier and Squarer for a Class of Finite Fields", IEEE Transactions on Computers, vol. 51, p. 521-529, 2002

[118] Montgomery P.L., "Modular Multiplication without Trial Division", Math. Computation, vol. 44, p. 519-521, 1985

[119] Tenca A.F. and Koc C.K., "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm", IEEE Transactions on Computers, vol. 52, p. 1215-1221, 2003

[120] Walter C., "Technique for the Hardware Implementation of Modular Multiplication", Proceedings of Second IMACS International Conference on Circuits, Systems & Computers, Athens, vol. 2, p. 945-949, 1998

[121] Savaş E., Tenca A. F., and Koç C. K., "A scalable and unified multiplier architecture for finite fields GF(*p*) and GF(2^*m*")", Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, Springer-Verlag, LNCS, vol. 1965, p. 277-292, 2000

[122] Savaş E., Tenca A. F., Ciftcibasi M. E., and Koç. C. K., "Novel multiplier architectures for GF(*p*) and GF(2^*n*)", Proceedings of Computers and Digital Techniques, vol. 151, p. 147-160, 2004.

[123] Groβschädl J., "A bit-serial unified multiplier architecture for finite fields GF(*p*) and GF(2^m)", Proceedings of CHES 2001, Paris, p. 202-218, 2001

[124] Wolkerstorfer J., "Dual-Field Arithmetic Unit for GF(*p*) and GF(2^m)", Proceedings CHES 2002, San Francisco, vol. 2523, p. 500-514, 2002

[125] Savaş E. and Koç. C. K., "Architectures for unified field inversion with applications in elliptic curve cryptography". Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems - ICECS 2002, Dubrovnik, Croatia, vol. 3, p. 1155-1158, 2002

[126] Gutub A. A.-A., Tenca A. F., Savaş E. and Koç., C. K. "Scalable and unified hardware to compute Montgomery inverse in GF(*p*) and GF(2^*n*)", Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, Springer-Verlag, LNCS, vol. 2523, p. 484-499, 2002.

[127] Au L.S. and Burgess N., "A (4:2) Adder for Unified GF(*p*) and GF(2^n) Galois Field Multipliers", Proceedings of 36th Asilomar Conference on Signals, Systems and Computers, vol. 2, p. 1619-1623, 2002

[128] Azivienis A., "Signed-Digit Number Representations for fast parallel arithmetic", IRE Transactions on Electronic Computers, vol. 10, p. 389-400, 1961

[129] Takagi N., Yasura H. and Yajima S., "High-speed VLSI multiplication algorithm with a redundant binary addition tree", IEEE Transactions on Computers, vol. 34, p. 789-796, 1985

[130] Santoro M.R. and Horowitz M.A., "SPIM: a pipelined 64×64-bit iterative multiplier", IEEE Journal of Solid-State Circuits, Vol. 24, p. 487-493, 1989

[131] Sutherland I., Sproull B. and Harris C. "Logical Effort – Designing Fast CMOS Circuits", Morgan Kaufmann Publishers, 1999

[132] Ho R., Hai K. W. and Horowitz M., "The Future of Wires", Proceedings of the IEEE, vol. 89, p. 490-504, 2001

[133] Knuth D. E., "The Art of Computer Programming, Vol. 2", Seminumerical Algorithms, 2nd edition, Addison-Wesley, Reading, Mass., 1981

[134] Bosselaers A., Govaerts R. and Vandewalle J. "Comparison of three modular reduction functions", Crypto '93, Springer-Verlag, LNCS, vol. 773, p. 175-176, 1993

[135] Dhem J–F., "Design of an efficient public-key cryptographic library of RISC-based smart cards", PhD thesis, Université catholique de Louvain – UCL Crypto Group – Laboratoire de microélectronique (DICE), 1998

[136] Brickell E.F., "A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography", Proceedings of Crypto'82, Plenum Press, p. 51-60, 1983

[137] Barratt P., "Communications authentication and security using public key encryption", Master's thesis, Oxford University, 1984

[138] Barrett P. "Implementing the Rivest, Shamir and Adleman public key encryption algorithm on standard digital signal processor" In Advances in Cryptology – CRYPTO '86, Santa Barbara, California, LNCS, vol. 263, p. 311-323, 1987

[139] Großschädl J., "High-Speed RSA Hardware Based on Barett'S Modular Reduction Method", CHES 2000, Springer-Verlag Berlin Heidelberg 2000, LNCS, vol. 1965, p. 191-203, 2000

[140] Savas E., "Implementation Aspects of Elliptic Curve Cryptography", Ph.D. Thesis, Department of Electrical & Computer Engineering, Oregon State University, 2000
http://islab.oregonstate.edu/papers/00Savas.pdf

[141] Tenca A. F. and Koc C. K. "A scalable architecture for Montgomery multiplication", Cryptographic Hardware and Embedded Systems, First International Workshop, Worcester, MA, USA, Springer-Verlag, LNCS, vol. 1717, p. 94-108, 1999

[142] Drábek V., "Montgomery Multiplication in GF(*p*) and GF(2^*n*)", Proceedings of Electronic Devices and Systems, Brno, CZ, p. 106-109, 2003

[143] Tenca A. F. and Koc. C. K., "A scalable architecture for modular multiplication based on Montgomery's algorithm", IEEE Transactions on Computers, vol. 52, p. 1215-1221, 2003

[144] Wu H., "Montgomery Multiplier and Squarer in GF(2^m)", Proceedings of Cryptographic Hardware and Embedded Systems (CHES 2000), vol. 1965, p. 264-276, 2000

[145] Koc C. K. and Acar T. "Montgomery Multiplication in GF(2^k)" Design, Codes and Cryptography, Kluwer Academic Publishers, Boston, vol. 1, p. 57-69, 1998

[146] Hong J-H, Wu C-W, "Radix-4 modular multiplication and exponentiation algorithms for the RSA public-key cryptosystem", Proceedings of the 2000 conference on Asia South Pacific design automation, Yokohama, Japan, p. 565-570, 2000

[147] Kornerup P., "High-radix modular multiplication for cryptosystems," Proceedings of 11th IEEE Symposium Computer Arithmetic, Windsor, Ontario, Canada, p. 277-283, 1993.

[148] Tenca A. F., Todorov G. and Koç C. K., "High-Radix Design of a Scalable Modular Multiplier", Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems, Spinger-Verlag, vol. 2162, p.185-201, 2001

[149] Takagi N., "A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation", IEEE Transactions on Computers, Vol. 41, p. 949-956, 1992

[150] Walter C. D., "Space/Time Trade-Offs for Higher Radix Modular Multiplication Using Repeated Addition", IEEE Transactions on Computers, vol. 46, p139-141, 1997

[151] Groβschädl J., "A unified radix-4 partial product generator for integers and binary polynomial", Proceedings of the 35th IEEE International Symposium on Circuits and Systems (ISCAS 2002), Vol. 3, p. 567-570, 2002.

[152] Booth A. D. "A Signed Binary Multiplication technique" Quarterly J. Mechanics and Applied Mathematics, vol. 4, p. 236-240, 1951

[153] Shand M. and Vuillemin J. E., "Fast implementations of RSA cryptography", Proceedings of the 11th IEEE Symposium on Computer Arithmetic, Windsor, Canada, p. 252-259, 1993

[154] Orup H., "Simplifying Quotient Determination in High-Radix Modular Multiplication", Proceedings of 12th Symposium Computer Arithmetic, p. 193-199, 1995
file:///vlsi_soft/vlsi03/literature/Orup-HighRadix-ARITH95.pdf.

[155] Orlando G. and Paar C., “A scalable GF(p) elliptic curve processor architecture for programmable hardware.” Workshop on Cryptographic Hardware and Embedded Systems (CHES '01), vol. 2162, p. 348-363, 2001

[156] Daly A. and Marnane W. P., “Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic”, Proceedings of the 2002 ACM/SIGDA 10th international symposium on Field-progrmmable Gate Arrays, p. 40-49, 2002

[157] Knuth D. E., “The Art of Computer Programming, Vol. 2”, Seminumerical Algorithms, 2nd edition, Addison-Wesley, Reading, Mass., 1981

[158] Menezes A., van Oorschot P. and Vanstone S. “Handbook of Applied Cryptography” CRC Press, 1996
www.cacr.math.uwaterloo.ca/ha

[159] Sproull R. F. and Sutherland I. E., “Logical Effort: designing for speed on the back of an envelope”, IEEE Advanced Research in VLSI, Ed. C. Sequin, (Boston, MA: MIT Press), 1991

[160] Au L-S. and Burgess N., “Unified Radix-4 Multiplier for GF(*p*) and GF(2^*n*)”, Proceedings of 14th IEEE International Conference on Application-Specific Systems, Architectures and Processors, The Hague, Netherlands, p. 226-236 2003

[161] Biham E., “A Fast New DES Implementation in Software Source”, Proceedings of the 4th International Workshop on Fast Software Encryption, Springer-Verlah, LNCS, Vol. 1267, p. 260-272, 1997

[162] Clavier C., Coron J-S. and Dabbous N., “Differential Power Analysis in the presence of hardware countermeasures”, Proceedings of Cryptographic Hardware and Embedded Systems – CHES 2000, LNCS, vol. 1965, p. 252-263, 2000

[163] Hewlett Packard Company, “A Fast Implementation of DES and Triple-DES on PA-RISC 2.0, 2000
http://www.usenix.org/events/osdi2000/fulll_papers/corella.pdf

[164] Eldridge S. and Walter C., “Hardware Implementation of Montgomery’s Modular Multiplication Algorithm”, IEEE Transactions on Computers, vol. 42, p. 693-699, 1993