Efficient Implementation of Elliptic Curve Cryptography on FPGAs

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Jamshid Shokrollahi

aus

Tehran, Iran

Bonn 2006

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Referent: Prof. Dr. Joachim von zur Gathen

2. Referent: Prof. Dr. Ulrich Rückert (Universität Paderborn)

Tag der Promotion: 18.12.2006

Erscheinungsjahr: 2007

Diese Dissertation ist auf dem Hochschulschriftenserver der ULB Bonn http://hss.ulb.uni-bonn.de/diss_online elektronisch publiziert.

I would like to thank my supervisor Prof. Dr. Joachim von zur Gathen for giving me the opportunity of doing a PhD, for his support, and for teaching me how to work efficiently. Gratitude goes also to my co-referent Prof. Dr. Ulrich Rückert and the other members of my committee Prof. Dr. Jens Vygen und Prof. Dr. Michael Clausen.

I would also like to thank Courtney Tenz and Jeff Godden for their proofreadings.

Contents

Chapter 1

Introduction: Cryptography and Hardware

In the past traditional communications were based on letters, payments were done using checks or cash, and secret documents were saved in sealed boxes. Today everything is changed, and is changing quickly. Everyday more people buy cell phones, the number of e-mail users goes up, and more people pay their payments over the internet. Paperless office strategies save and process documents in electronic format. These trends are going to make the life easier but at the same time produce security risks. Traditional paper-based systems have been developed during a long time, in parallel to suitable laws for their security and reliability. The rapid development of electronic communication systems requires a secure infrastructure, too. Cryptography is the mathematical tool which is used by security engineers to secure data against unauthorized access or manipulation. Cryptography supplies the people, who are responsible for security, the required utilities to hide data, control accesses to them, verify their integrity, and estimate the required cost and time to break the security.

Like every other useful service, security will not be achieved for free. Implementing cryptography tasks costs time, money, and energy. The focus of this work is about the design of an FPGA-based[1] elliptic curve cryptography co-processor (ECCo) and the study of different techniques which can be used to increase its performance. Such a co-processor can influence applications in different ways: By increasing the speed, it enables more people to use the system in the same time and increases the availability. It can reduce the overall system costs. If energy consumption is minimized, this processor can decrease the total energy, and for example increase the battery lifetime in cell phones. Such improvements can be done in different levels as we see in Chapter 2. Implementing a fast co-processor, in this work, is done by studying the well-known methods in different areas. But the proposed novel improvements concern finite field multiplication only. This task is at the root of elliptic curve cryptography and every improvement in that can influence directly the performance of the co-processor. Finite fields of characteristic 2 are specially attractive for hardware designers since computation in these fields does not produce a carry, which contributes to long and complicated paths in hardware designs. It is the main reason that we study such fields.

There are two popular kinds of cryptographic protocols, namely public key and private key protocols. In private key protocols, a common key is used by both communication partners and for both encryption and decryption. Among them are DES, IDEA, and AES. These systems provide high speed but have the drawback that a common key must be established for each pair of participants. In public key protocols we have two keys, one is kept private and used either for decryption (confidentiality) or encryption (signature) of messages. The other key, the public key, is published to be used for the reverse operation. RSA, ElGamal, and DSS are examples of public key systems. These systems are slower than the symmetric ones, but they provide arbitrarily high levels of security and do not require an initial private key exchange. In real applications, both types are used. The public key algorithm first establishes a common private key over an insecure channel. Then the symmetric system is used for secure communication with high throughput. When this key expires after some time, a new key is established via the public key algorithm again.

[1] Field Programmable Gate Array

Due to the comparative slowness of the public key algorithms, dedicated hardware support is desirable. In the second chapter of this work, we present different structures for FPGA-based implementations of a cryptographic co-processor using elliptic curves. Then we will present some results about efficient finite field arithmetic which can be used to improve the performance of such processors. FPGA-based cryptography co-processors avoid a series of drawbacks of ASIC[2] based systems:

- A cryptography algorithm is secure as long as no effective attack is found. If this happens, the algorithm must be replaced. FPGAs facilitate a fast and cost effective way of exchanging the algorithm, in particular of switching to a higher key length.
- In electronic commerce servers, cryptographic algorithms can be exchanged often for the purpose of adaption to the current workload, depending on the type of cryptography that is mainly used (public key or symmetric key). This can be done by exploiting the FPGAs reconfigurability.
- Elliptic curve cryptosystems possess several degrees of freedom like Galois field characteristic, extension degree, elliptic curve parameters, or the fixed point generating the working subgroup on the curve. FPGAs allow for an effortless adaption to changing security or workload requirements.
- The empirical results of testing various approaches on an FPGA may later be of help in designing an efficient ASIC, where such experiments would be much more costly.

1.1 Related Works and Document Structure

The contributions of the present work can be summarized in the following items:

- The comparison of the costs of polynomial and normal basis arithmetic in two-input and FPGA models in Section 2.2.

[2] Application-Specific Integrated Circuit

- Analyzing the effect of different point representations on the performance of parallel implementations of elliptic curve cryptography over fields of characteristic 2 in Sections 2.3 and 2.4.

- Implementing a very fast FPGA-based ECCo using parallel arithmetic units in Section 2.5.

- Analyzing combinations of different recursive polynomial multiplications to reduce the area requirements of hardware implementations in Section 3.3.

- Decreasing the latency of pipelined recursive polynomial multipliers by decreasing the recursion degree in Section 3.4.

- Introducing a new structure for efficient changing between polynomial representations and optimal normal bases of type II in special finite fields. This technique which is introduced in Chapter 4 results in efficient normal basis multipliers which are analyzed in that chapter.

Due to the importance of elliptic curve cryptography, there are a lot of publications in this area. The following paragraphs describe the document structure together with the most important publications related to each chapter.

Chapter 1, this chapter, is the opening of the work and contains pointers to references for further information. It begins with a very short introduction to cryptography and the group of points on an elliptic curve, and is continued with an overview of the structure of the specific FPGAs which are used. These topics are followed with the definitions of the cost parameters which are considered when designing the circuits. Finally this chapter is concluded with some possible applications where the results of this work can be applied. A sample application, a PCI-based cryptography co-processor, has been implemented and the benchmarks are presented. It should be mentioned, that the materials in this chapter are in no way, a complete text book about cryptography or FPGAs. We assume, that the reader is familiar with finite fields and basic hardware methods like pipelining.

Chapter 2 describes the steps of the design and implementation of an elliptic curve coprocessor (ECCo). The ECCo should be optimized to have small area. Comparisons have been performed between multipliers which can be adapted to tight area constraints. Since the target platforms are FPGAs, implementation costs have been compared in classical circuit analysis models and other models which are closer to the structure of the FPGAs used. Some of the algorithms use a particular representation of points on an elliptic curve called "mixed coordinates". There are some computations considering the mixed coordinates when fields of characteristic 2 are used. These results can be derived from the works of López & Dahab (1999b) and Cohen *et al.* (1998). Materials of this chapter which are the results of cooperation with the working group AGTeich are already published in Bednara *et al.* (2002a) and Bednara *et al.* (2002b). There are several other works which describe the application of FPGAs for elliptic curve curve cryptography or finite field arithmetic (see Gao *et al.* (1999), Gregory *et al.* (1999), Leong & Leung (2002), Orlando & Paar (1999) and Lutz & Hasan (2004)). The distinguishing factor in our work is the application of parallelism in both bit and finite field operations. As we will see in Chapter 2, the area and time costs of finite field multipliers grow faster than linear when the number of output-bits per clock-cycle is increased. This shows that it is always better to use as many small parallel multipliers as possible instead of using a single multiplier with a large number of output bits per clock cycle. Unfortunately the performance of the FPGA-based systems depends on the platform used and a direct comparison is possible only when considering the same target FPGA. From the above implementations the only comparable work belongs to Lutz & Hasan (2004) which requires 0.233 ms for a point multiplication on a generic curve over $\mathbb{F}_{2^{163}}$, when a clock frequency of 66 MHz is used. Our design on the other hand requires 0.18 ms for a generic curve over $\mathbb{F}_{2^{191}}$ with the same clock frequency and on the same FPGA. It should be pointed out that their design is optimized for the Koblitz curves (see Hankerson *et al.* (2003)) and not generic cases.

Chapter 3 can be considered the most important part of this thesis. It contains results about applications of asymptotically fast multiplication in hardware. These methods have been known for a long time but their high crossover points in software did not let designers enjoy their high performance in practical situations. Software implementations of the Karatsuba multipliers using general purpose processors have been discussed thoroughly in the literature (see Paar (1994), Bailey & Paar (1998), Koç & Erdem (2002), Hankerson *et al.* (2003), Chapter 2, and von zur Gathen & Gerhard (2003), Chapter 8). There are, on the contrary, only few publications about the hardware implementations. Jung *et al.* (2002) and Weimerskirch & Paar (2003) suggest the use of algorithms with $O(n^2)$ operations to multiply polynomials which contain a prime number of bits. The number of bit operations is, by a constant factor, smaller than the classical method and yet asymptotically larger than those for the Karatsuba method. Grabbe *et al.* (2003a) propose a hybrid implementation of the Karatsuba method which reduces the latency by pipelining and by mixing sequential and combinational circuits. The goal of this chapter is to present a method to decrease the resource usage of polynomial multipliers by means of both known algorithmic and platform dependent methods. This is achieved by computing the best choice of hybrid multiplication algorithms which multiply polynomials with at most 8192 bits using six recursive methods, namely: classical, Karatsuba, a variant of Karatsuba for quadratic polynomials, and three methods of Montgomery (2005) for polynomials of degrees 4, 5, and 6, respectively. In addition to the above algorithmic, or machine independent optimization we use a second type of optimization, which is machine-dependent, to design a 240-bit multiplier with small area-time cost. This 240-bit multiplier covers in particular the 233-bit polynomials proposed by NIST for elliptic curve cryptography (FIPS PUB 186-2 (2000)). Many of the materials of this chapter are new results and some of them are published in Grabbe *et al.* (2003a), von zur Gathen & Shokrollahi (2005), and von zur Gathen & Shokrollahi (2006). For example, finding the optimum hybrid limits, decreasing the number of recursive stages, and the code generator.

Chapter 4 describes the use of sub-quadratic multiplication methods for normal basis arithmetic in finite fields. Amin Shokrollahi initiated the discoveries in this chapter. Normal bases are popularized in finite fields because of the ease of squaring but they have the drawback that multiplication in these bases is more expensive than in polynomial bases. Multiplication in normal bases of small type has important applications in cryptography, so that most of cryptography standards suggest the use of finite fields which contain such bases (see FIPS PUB 186-2 (2000)). There are several works detailing the implementation of these multiplications, starting with Omura & Massey (1986) which introduced the Massey-Omura multiplier. Mullin *et al.* (1989) define optimal normal bases, which minimize the area and the time complexities of this multiplier and Gao & Lenstra (1992) specify exactly the finite fields for which optimal normal bases exist. Following these works there are several proposals for the efficient multiplications using optimal normal bases and especially those of type 2. The parallel Massey-Omura multiplier for $\mathbb{F}_{2^n}$ can be implemented, with at least $n(3n-2)$ gates, whereas multiplications of polynomials of degree $n-1$ is done, classically, using $2n^2-2n+1$ gates. Sunar & Koç (2001) and Reyhani-Masoleh & Hasan (2002) decrease the cost of type 2 multiplication to $n(5n-1)/2$ by suitably modifying the Massey-Omura multiplier. Gao *et al.* (2000), on the other hand, decrease the multiplication cost in optimal normal bases of type 2, asymptotically, to $2\mathsf{M}(n)$, where $\mathsf{M}(n)$ is the cost of multiplying two polynomials of degree $n-1$ (of length n). This allows the application of asymptotically fast polynomial multiplication methods for normal bases as well. The structure reported in Chapter 3 decreases this cost asymptotically to $\mathsf{M}(n)+O(n\log n)$ by the addition of a suitable small size circuit to a polynomial multiplier. This small circuit is used to convert from the normal basis to an appropriate polynomial representation. A comparison of the area of this multiplier with the other proposed architectures in the literature shows its suitability for small area implementations. Results of this chapter can also be used for efficient change of basis between the polynomial and the normal bases as a mechanism against side-channel attacks (see Park *et al.* (2003)). Chapter 5 summarizes the results of this work.

1.2 Cryptography

In this section we describe the two kinds of cryptography systems, namely public and private key systems. The results of this work can be used in cryptography systems but are not directly cryptographical results. Hence, we avoid formal definitions and limit ourselves to brief explanations which are sufficient to represent applications of this work.

1.2.1 Private Key Cryptography

Almost all cryptographic protocols are based on the same principle. They contain a function which, by means of a parameter called the encryption key, can be easily computed. The inverse of this function is hard to compute unless a trapdoor function (a second key corresponding to the former one) is known. A general assumption made during the analysis of the security of a system is that all information about the system except the trapdoor key are known by the adversary. The previously mentioned group of public and private key systems are based on the way these keys are generated and kept.

In a private key system encryption and decryption are done using the same key which should be kept secret, otherwise the system is broken. Figure 1.1 shows a scenario where communication is secured via a private key system. Here Eve does not know the private key and cannot get any information even if she has access to the channel.

There are several private key algorithms like Rijndael (AES) and 3DES. Private key systems are generally characterized by very high performance. But they cannot normally be used alone. Their applications will be completed with public key cryptosystems which are introduced in Diffie & Hellman (1976).

1.2.2 Public Key Cryptography

As we have already mentioned private key systems are generally very efficient but there is the need for other kinds of cryptosystems in practice. Consider as an example the setup in Figure 1.1. Alice and Bob have never met each other and their only connection

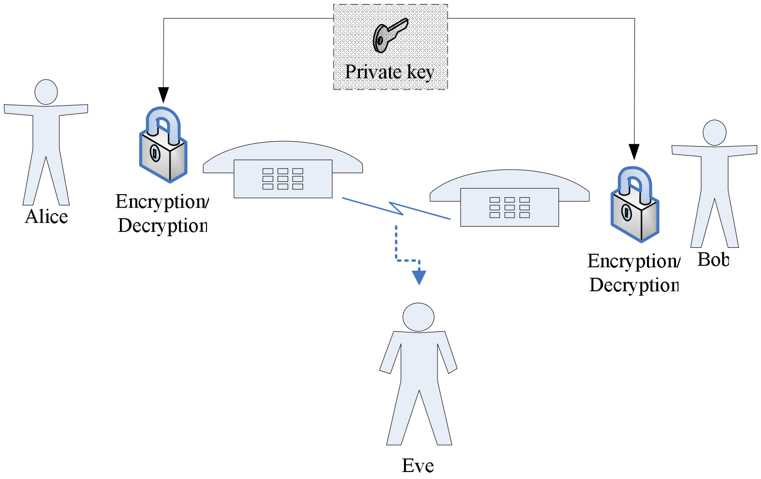


Figure 1.1: A private key cryptography scenario

is a channel which is accessible to Eve. In this case they have never the possibility of establishing a common secret key using private key cryptosystems only. As another case consider the scenario in which instead of Alice and Bob, a group of 1000 people want to communicate with each other. In this case every user requires 999 keys and the overall system requires 999000 keys to be generated.

In public key cryptosystems encryption and decryption are done using two different keys. One of the keys is published and the other is kept secret. When one party is going to sign a message the encryption key is kept secret but the key to verify the signature will be published. On the other hand when a secret message is to be sent the encryption key will be published while the key to open the message will be kept secret by the owner.

Figure 1.2 is an example for a public key system where the information should be kept secret during transmission. In this system messages sent to a user are encrypted by his encryption system and he is the only person who has access to the corresponding private key and can decrypt the message.

There are several types of public key cryptosystems. A major group of these systems is based on the difficulty of solving the discrete logarithm problem or DLP for short. In the next section we explain the elliptic curve variant of this problem.

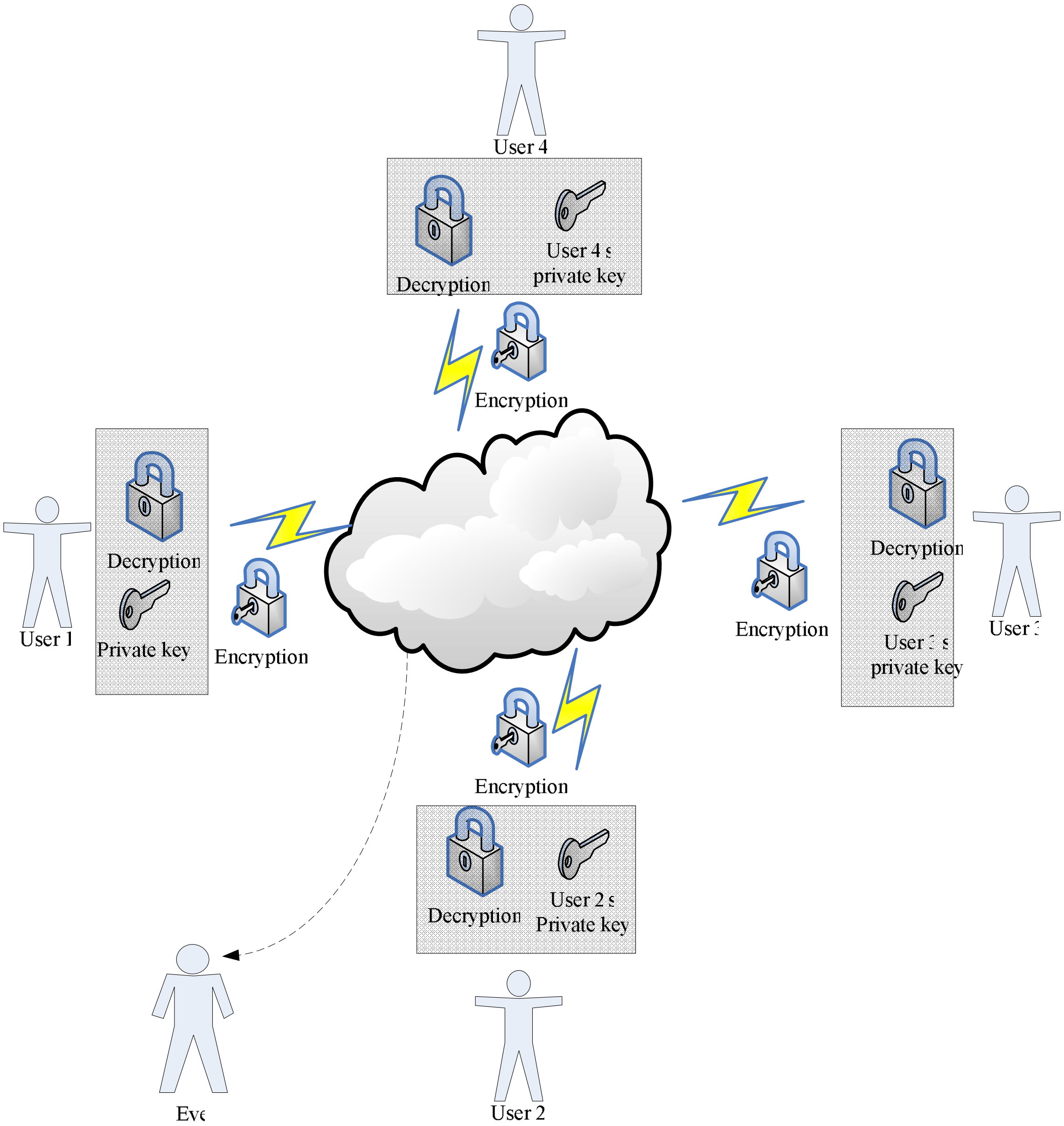


Figure 1.2: A public key cryptography scenario

1.2.3 Elliptic Curves and the Discrete Logarithm Problem

Let E be an elliptic curve defined, in the affine version, by the Weierstrass equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

which is defined over a finite field K. It can be shown, that there is a group associated with the points on this curve (see Silverman (1986), Chapter III, Section 2, Page 55 for the proof). The operation of this group, the addition of points, is defined in a special manner which is shown in Figure 1.3. Let $\mathcal{S}$ and $\mathcal{Q}$, in the part (a) of that figure, be two distinct points on an elliptic curve. There is a straight line through these points which intersects the curve in another third point, $-\mathcal{R}$ in that figure. The mirror of $-\mathcal{R}$ with respect to the x-axis is a new point, $\mathcal{R}$, which is defined as the sum of $\mathcal{S}$ and $\mathcal{Q}$. When a point is added to itself the tangent line at that point is used instead, as shown in Figure 1.3-b. Like the last case, the sum is computed as the mirror of the next intersection with respect to the x-axis. As a common precept in group theory, here a zero element is needed. It can be easily verified, that if the straight line through a point is parallel to the y-axis, it intersects the curve in the mirror of the original point with respect to the x-axis. Mirroring this point results in the original point. The zero point $\mathcal{O}$ is virtually defined to be in the infinity on the y-axis to achieve a line which is parallel to the y-axis for every point on the curve. This point is generally called the “point at infinity”.

Now that we can add two points, distinct or equal, we can compute any integer multiple of a point. We call this operation the “point multiplication”. In this way $n\mathcal{Q}$ is the point which is computed by $n - 1$ times addition of the point $\mathcal{Q}$ to itself. Since the set of points generate a group this product is well defined and does not depend on the way the points are added together. The aim of our co-processor is to compute $n\mathcal{Q}$ for a given $\mathcal{Q}$ and an integer n, when the elliptic curve is already specified.

The DLP on elliptic curves is the problem of computing n from $\mathcal{Q}$ and $n\mathcal{Q}$. It is generally assumed that, at least for general enough curves, this cannot be solved in polynomial time, i.e., in a number of operations which is expressible in a polynomial of the bit-size

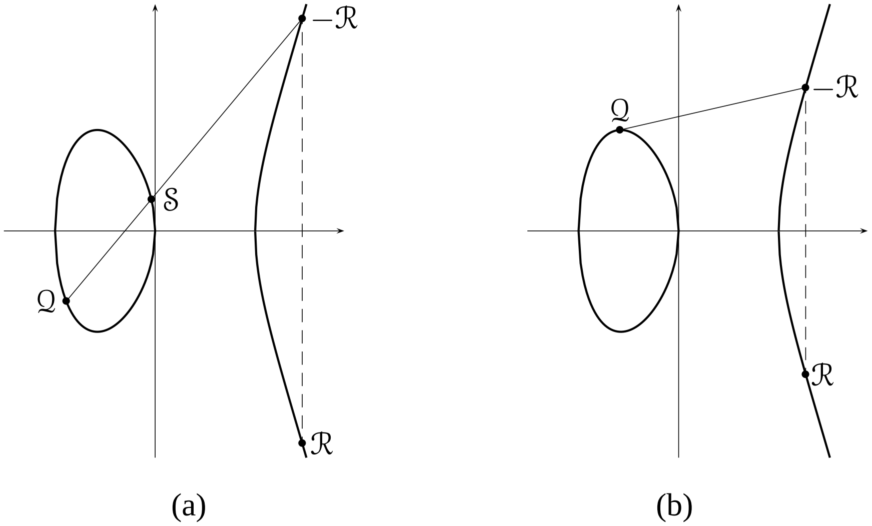


Figure 1.3: (a) Addition and (b) doubling of points on an elliptic curve

of the finite field, i.e., $\log_2 \#F$. It should be pointed out that for some very special elliptic curves the DLP is known to be easy (see Blake *et al.* (1999), Chapter III, Section 3.2, Page 37). We assume that the given finite field and the curve are not of this form. Our elliptic curves, for fields of characteristic 2, are of the general form:

$$E : y^2 + xy = x^3 + ax^2 + b,$$

with $a, b \in \mathbb{F}_{2^n}$, $b \neq 0$.

To show where and how this project can be used, we describe some applications of elliptic curve cryptography and how using an elliptic curve co-processor can improve the performance of the system.

1.2.4 Applications

Key Establishment

Consider again the scenario presented in Figure 1.1. As we have already mentioned, if Alice and Bob have never met each other they cannot agree upon a secure and common private key. Even if they establish a key and later doubt the security of this key (for example if they find out Eve could recover some or all of bits of the key) they cannot change the key unless they have a secure channel or meet each other. A solution to the key establishment problem has been suggested for the first time by Diffie & Hellman (1976). This situation which is shown in Figure 1.4 makes use of the difficulty of solving the DLP. It is assumed that Alice and Bob have already selected an elliptic curve and a point $\mathcal{Q}$ on it. The order of the group of points, n, is already known .

1 Alice selects a random number $1 < r < n$, computes $r\mathcal{Q}$, and sends $r\mathcal{Q}$ to Bob.

2 Bob selects a random number $1 < s < n$, computes $s\mathcal{Q}$, and sends $s\mathcal{Q}$ to Bob.

3 Alice and Bob use $rs\mathcal{Q}$ as the common secret key for secure communication using the private key system.

Figure 1.4: The Diffie-Hellman key establishment protocol using elliptic curves.

As we see Eve's task should be computing $rs\mathcal{Q}$ from $r\mathcal{Q}$ and $s\mathcal{Q}$. If the DLP were easy to solve Eve could find r and s by observing the communication. But she could probably solve her problem even without solving the DLP. It is conjectured that her task is as hard as solving the DLP but, despite numerous efforts to prove this assertion, the general case is still open (see e.g. Maurer (1994)).

Here all required operations except finding random numbers are multiplications on elliptic curves which shows how useful an elliptic curve co-processor can be for this application.

Digital Signatures

As another scenario consider a situation, where Bob receives a message from Alice. For example a message that the key has been lost and a new session key has to be established. How can Bob be sure that this message is from Alice? Could it not be the case that Eve wants to completely redirect Bob's communication with Alice to herself?

A public key protocol has been suggested by ElGamal (1985), based on which the digital signature standard (or DSS for short) has been proposed (see FIPS PUB 186-2 (2000)). Algorithms for signing and signature verifications in elliptic curve counterparts of this scenario (ECDSA) are shown in Algorithms 1 and 2 respectively. The function H in these algorithms is some secure hash algorithm (FIPS recommends SHA); we do not discuss security of hash functions here. For us at the moment, it is a function that takes a sequence of bits and outputs a sequence of fixed length, say 160 bits, with some specific properties (see FIPS PUB 180-1 (1993) for more information).

Algorithm 1 Message signing in ECDSA

Input: An elliptic curve with a fixed point $\mathcal{Q}$ on it, together with its order n, the private key $1 < d < n - 1$, the public key $\mathcal{R} = d\mathcal{Q}$, and the message m to be signed.

Output: The pair of integers (r, s) as the signature of the message m.

```
 1: Select a random integer $1 < k < n - 1$
 2: Compute $k\mathcal{Q} = (x_1, y_1)$ and $r = x_1 \mod n$
 3: **if** $r = 0$ **then**
 4:     Go to 1
 5: **end if**
 6: Compute $k^{-1} \mod n$
 7: Compute $s = k^{-1}(H(m) + dr) \mod n$
 8: **if** $s = 0$ **then**
 9:     Go to 1
10: **end if**
11: **return** $(r, s)$
```

Here we see that the key generation has one elliptic curve multiplication and the signing and verification phases require one and two multiplications respectively. These are operations which can be accelerated using elliptic curve co-processors.

Algorithm 2 Signature verification in ECDSA.

Input: An elliptic curve with a fixed point $\mathcal{Q}$ on it, together with its order n, the public key $\mathcal{R} = d\mathcal{Q}$, the message m which is signed, and a pair of integers (r, s) as the signature.

Output: TRUE if (r, s) is a valid signature for m, FALSE otherwise.

```
Compute $c = s^{-1} \mod n$ and $H(m)$
Compute $u_1 = H(m) \cdot c \mod n$ and $u_2 = r \cdot c \mod n$
Compute $u_1\mathcal{Q} + u_2\mathcal{R} = (x_0, y_0)$ and $v = x_0 \mod n$
if $r = v$ then
    Output TRUE
else
    Output FALSE
end if
```

1.3 Hardware for Cryptography

In the last section we saw where elliptic curve cryptography can be used. But is it really necessary to build a special co-processor for it or all of our problems can be solved using current processors to perform algorithms? In this section we consider two special cases where co-processors can have important advantages which can not be achieved by only using general purpose microprocessors.

1.3.1 Smart Cards

Smart cards are going to be a part of our life. A lot of our applications are done using smart cards. Identifying ourselves in a mobile network is done using SIM cards (Subscriber Identity Module). We use smart cards as insurance cards, bank cards, and in several other applications. These are some chips with limited amounts of memory and small general purpose processors. Implementations of cryptographic algorithms on these processors are generally slow and require several operations but can be reduced to fewer ones when special purpose co-processors are used. These reductions save energy and time.

Another possibility is to extend the smart card microprocessor with some special arithmetic modules. Results which are gathered in this project can be used in each of these strategies.

1.3.2 Accelerator Cards

Another situation where a crypto co-processor can be useful is in e-commerce servers. In these applications the computational power is not so limited as in smart cards but there are several requests which should be responded to simultaneously. In an e-commerce server several users try to connect to a server and send requests for which a signature must be generated or verified. At the same time users, who are already connected, send and receive information which should be encrypted. The processor is here not only responsible for cryptographic algorithms but it should also process some other tasks like network operations which are assigned to every server. Equipping a server with a cryptography accelerator card will help the main microprocessor to concentrate on server operations. Otherwise each user would face a long waiting delay for his jobs to be done.

1.3.3 FPGA

FPGAs or field programmable gate arrays are valuable tools which can help in several design stages. On the one hand an FPGA module can be used to develop a prototyping model. Developing an ASIC chip is very expensive because once a design is finished, changing its structure requires a completely new chip. FPGAs give designers the opportunity to test the complete hardware (up to some timing limitations) for possible bugs and problems.

On the other hand with the development of large and inexpensive FPGAs it is possible to design the complete system in a single chip (an SoC, or a system on chip). These systems perform all necessary operations and can be reconfigured at any time. A system

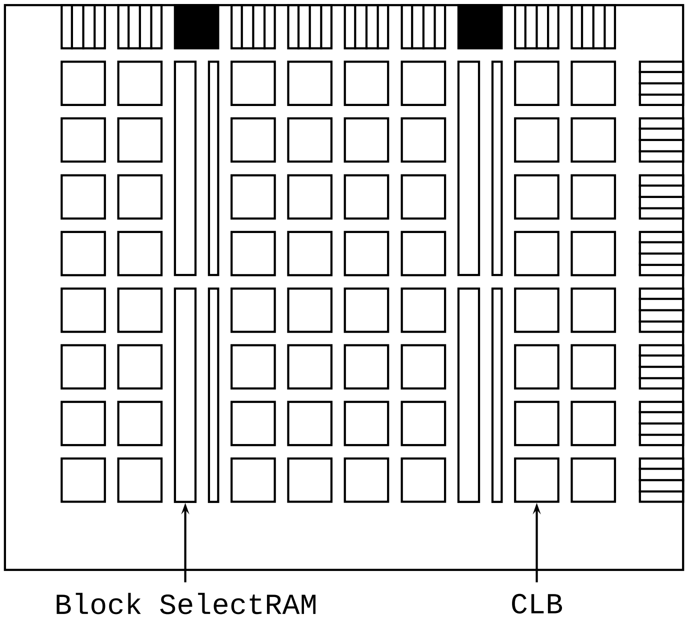


Figure 1.5: A simplified version of a Virtex-II FPGA

which is developed and encounters a problem needs only to be reconfigured to solve the problem. For our example with the accelerator card it is possible to make the co-processor on an FPGA and modify it with respect to the workload during the operation.

The designs explained later, in Chapters 2 and 3, are implemented on the FPGAs from Xilinx company. A simplified overview of the structure of an FPGA in the Virtex II family, on which the designs are implemented, is shown in Figure 1.5. For complete information about these FPGAs see the online documentation on the internet (Xilinx 2005). There are several modules on such an FPGA, but we mention here only two of them which are important in our designs.

Block SelectRAM memory modules provide large 18 Kbit storage elements of dual-port RAM. These modules can be separately read and written by two processor modules and can be especially used as interfaces between processors and co-processors.

The Virtex-II configurable logic blocks (CLBs) are organized in an array and are used to build combinational and synchronous logic designs. Each CLB element is tied to a

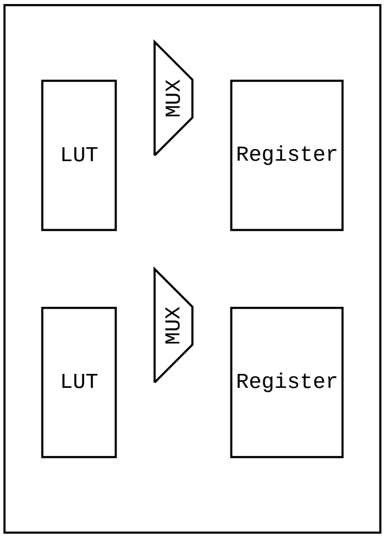


Figure 1.6: A simplified view of a single slice in a CLB of a Virtex-II FPGA

switch matrix to access the general routing matrix. A CLB element comprises 4 similar slices, with fast local feedbacks within the CLB. There are also fast connections between each CLB and its neighbors. Each slice includes several parts from which the most important ones for our designs are: two 4-input function generators, two single-bit D-type registers, and two multiplexers. The arrangement of these parts is shown in Figure 1.6. In this figure look-up tables (LUTs) are 4-input modules which have a single-bit output. These LUTs are each capable of implementing any arbitrarily defined boolean function of four inputs. The output of each LUT goes to the multiplexer and the register. The multiplexer selects, whether the LUT or the register should be connected to the output of the slice. This configuration is helpful when designing pipelined circuits.

1.3.4 Circuit Parameters

The cost parameters which we use to compare different designs are the implementation areas and the times required for the computation of results. We do not consider energy efficient implementation techniques and do not use the consumed energy as a cost function. The area of a combinational circuit – a circuit containing no memory element – is expressed as the number of two-input gates. In FPGA-based circuits this parameter can be compared with the number of LUTs since these blocks are responsible for the implementation of boolean functions in FPGAs. However, most of our designs use memory elements and are sequential. The pipelined multipliers in Chapter 3 especially use registers of the slices. To make a fair comparison between two different circuits in the case of sequential circuits, i.e., when timing and memory elements are important, we use the number of slices for the comparisons. In this way we count both the number of boolean function gates and the bit-registers.

The time parameter of a combinational circuit is computed as the depth of the circuit. This is the minimum allowable clock period, when this circuit is used without any further modifications. For the FPGA-based implementations it is better to compute the time cost as the product of the number of the clock cycles by the minimum allowable clock period. The latter contains several parameters like the propagation delays of cascaded LUTs, delay of routing resources including buffers in high fan-out nets, and setup times of the registers. For the case of two-input gate model the number of gates in the longest path represents the time cost.

The best method to compare two circuits is to analyze their area and time costs individually. But in some situations one parameter is more important (or more expensive) than the other. For example in a very small FPGA a much faster implementation which does not fit on the FPGA is of no use. Here the fair measure of comparison, which is also well established in the literature, is the product of area by time or AT. We use this measure to compare circuits when there is a conflict between the two parameters. The area-time measure has also another property which can be used for the comparison of parallel implementations of a method. Considering a circuit to be a parallel algorithm the area-time measure can be thought of as the consumed energy of that algorithm. Here the area is the sum of the power of processors which will be dissipated in the computation time. The energy of an ideal parallel implementation should be equal to that of a serial implementation, but there is often a penalty factor due to the parallelism. This measure

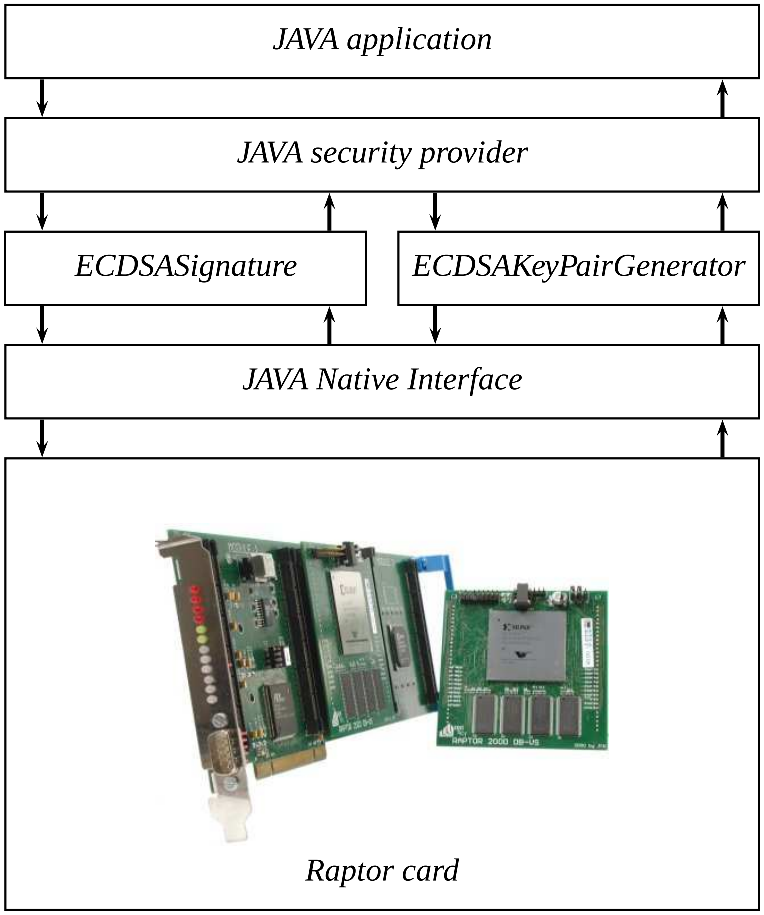


Figure 1.7: Using the raptor card as an ECDSA co-processor

shows how good different parallel implementations of a serial algorithm are.

1.3.5 A Typical Scenario, ECDSA Accelerator Card

As a typical scenario we have used our FPGA-based implementation to be on a PCI card in a PC. The system was designed to be JAVA compatible and developed in such a way that a programmer can access the processor functionalities through JAVA libraries. The platform which we used was the Rapid prototyping platform (Raptor card) from the working group AGRückert in the university of Paderborn. In the next section we describe the specifications of the system.

Digital Signatures in JAVA

The communication between JAVA applications and the ECCo is shown in Figure 1.7. The JAVA application starts by instantiating two objects of type `ECDSAKeyPairGenerator` and `ECDSASignature` which are derived from `DSAKeyPairGenerator` and `DSASignature` in the JAVA security provider respectively.

The class `DSAKeyPairGenerator` is a placeholder for classes which generate a set of public and private keys once a security parameter (generally the key length) and the algorithm are specified. In our implementation the security parameter, which specifies the extension degree of the finite field, can be only 191. To use other parameters the co-processor has to be synthesized again ,while the generation of the required VHDL-codes can be done automatically. The generated key pair is returned in a structure which is already defined by JAVA.

The class `DSASignature` contains virtual definitions of the necessary operations to perform digital signature algorithm, namely signing and verifying the signature. Again parameter passing is done in a standard way predefined by JAVA.

As we have already said these two classes contain only empty operations which have to be implemented for a cryptography system in JAVA. Our implementations perform the operations according to Algorithms 1 and 2. For the generation of a key pair only one multiplication over the elliptic curve is required which is done using the co-processor. There are several other operations like generation of random numbers, long integer arithmetic, and computing the SHA. These are performed using internal implementations of JAVA.

The security objects which we have implemented communicate with the card through Java Native Interface (or JNI). JNI is a facility which is put into JAVA systems to enable them to access libraries in other languages like the C language.

The driver for the card which is developed in the working group AGTeich of the University of Paderborn is able to get a 191-bit integer and a point Q, start the card to perform

Finite field	$\mathbb{F}_{2^{191}}$
Elliptic curve	$y^2 + xy = x^3 + ax + b$ $a = 1$ $b = 7BC86E2102902EC4D5890E8B6B4981$ $FF27E0482750FEFC03$
Number of points	$1569275433846670190958947355834614995815$ $261150867795429199 \cdot 4$
Key generation time	3.6 ms
Signing time	3 ms
Verification time	4 ms

Table 1.8: The specifications of our PCI based ECDSA co-processor with the timings achieved on a XCV2000e FPGA when the clock frequency is 12.5 MHz.

the point multiplication, and return the result. This driver which has been developed using C++ is a part of the system and is accessed through the JNI.

Some information about our design is shown in Table 1.8. In this table the parameter b is the hexadecimal representation of that element in $\mathbb{F}_{2^{191}}$. The best software based time known to us is about 3.5 ms using a 900 MHz UltraSPARC III processor [3] (see Gupta *et al.* (2004)). We know of no hardware implementation of ECDSA. The performance of our ECDSA co-processor can be increased by implementing long integer arithmetic in FPGA instead of using the JAVA inherent libraries. As it can be seen this system is fairly fast even with a very slow clock frequency. Embedding such a design in a handheld device can result in energy saving which is an important parameter.

[3] The time is not accurate since it has been visually interpolated from a continuous curve.

1.4 Conclusion

In this chapter, elliptic curve cryptography, the structure of FPGAs, and the parameters used to compare different hardware designs were briefly reviewed. The structure of a test elliptic curve digital signature (ECDSA) co-processor using an XCV2000e FPGA, has also been studied and the benchmarks have been presented.

Chapter 2

An FPGA-Based Elliptic Curve Cryptography Co-Processor

2.1 Introduction

Elliptic curve cryptosystems are public key protocols whose security is based on the conjectured difficulty of solving the discrete logarithm problem on an elliptic curve.

Assuming $\mathcal{Q}$ to be a point of order n on an elliptic curve it is desirable to compute $m\mathcal{Q}$, where m is an integer smaller than n. This will be done by using several additions, doublings, or possibly negations of points on the elliptic curve to achieve the result. These operations boil down to arithmetic operations in the finite field $K = \mathbb{F}_{q^n}$, over which the elliptic curve has been defined. In this work we concentrate on fields which have characteristic 2, i.e., q is a power of 2.

The required computations to compute $m\mathcal{Q}$ can be categorized at three levels. Each requires thorough investigations to enable the design of a high performance elliptic curve co-processor (see Figure 2.1):

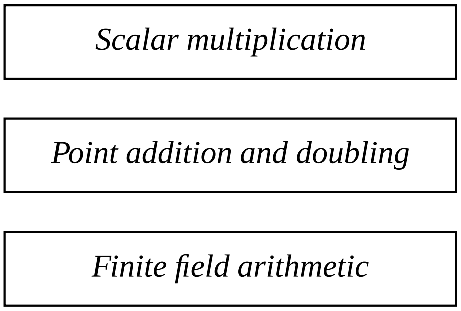


Figure 2.1: Three stages of performing elliptic curve point multiplication.

- **Scalar multiplication:** By scalar multiplication or point multiplication we mean the combination of additions and doublings of points to compute $m\mathcal{Q}$ for given m and $\mathcal{Q}$. There are several methods like the additive variant of repeated squaring or addition-subtraction chains which do this task using $O(\log m)$ doublings and additions (see Knuth (1998) and Morain & Olivos (1990)).

- **Point addition and doubling:** Multiplication of a point by a scalar consists of several additions, doublings, and possibly negations of points on the elliptic curve. Negation or computing $-\mathcal{Q}$ is almost free of cost but the other two operations are more expensive. There are several representations of points of an elliptic curve which influence point addition and doubling costs depending on the platform used.

- **Finite field arithmetic:** Point coordinates which have to be processed during point additions and doublings are elements of a finite field K. By accelerating operations in this field, we can improve the efficiency of point arithmetic and as an effect increase the performance of the co-processor. This can be done by optimal selection of finite field representations and by the hardware structures which perform addition, multiplication, and division in the field.

There are several published reports of efficient implementations of elliptic curve co-processors. see Gao *et al.* (1999), Gregory *et al.* (1999), Leong & Leung (2002), Orlando & Paar (1999), and Lutz & Hasan (2004)). The distinguishing factor in our work is the application of parallelism in both bit and finite field operations. Unfortunately the performance of the FPGA-based systems depends on the platforms and a direct comparison is possible only when the same target is used. Lutz & Hasan (2004) implemented their co-processor on the same FPGA model as used in this project. Their system requires 0.233 ms for a point multiplication on a generic curve over $\mathbb{F}_{2^{163}}$ when the clock frequency is 66 MHz. The current design on the other hand requires 0.18 ms for a generic curve over $\mathbb{F}_{2^{191}}$ with the same clock frequency and on the same FPGA. It should be pointed out that their design is optimized for Koblitz curves (see Hankerson *et al.* (2003)) and not generic curves.

This chapter is arranged in the following manner: Section 2.2 compares two popular finite field representations, namely the polynomial basis and the normal basis for the efficiency of arithmetic, when elliptic curves are implemented. Section 2.3 compares different representations of points and their effect on the efficiency when parallel and serial implementations are considered. Section 2.4 compares different methods of computing an integer multiple of a point. Section 2.5 presents the data-path and important modules in the implemented FPGA-based co-processor followed by the benchmarks achieved in Section 2.6. Finally Section 2.7 summarizes the results of the previous sections. Some of the materials of this chapter have been already published in Bednara *et al.* (2002a) and Bednara *et al.* (2002b).

2.2 Finite Field Arithmetic

It is known that the additive group of a finite field $\mathbb{F}_{q^n}$ can be represented as a vector space of degree n over $\mathbb{F}_q$. In this manner elements of $\mathbb{F}_{2^n}$ are represented by vectors of length n consisting of 0's and 1's which can be added using XOR operations. The operations of multiplication, squaring, and inversion depend highly on the selected basis.

There are three famous finite field representations, namely: polynomial, normal, and dual bases. Arithmetic in dual bases requires a change of representation for each operation. This makes these bases inefficient for cryptographic purposes because the finite fields which are used here are of significant size and conversion would be inefficient. We consider only the two other bases in this section.

2.2.1 Polynomial and Normal Bases

One popular representation for finite fields is the polynomial basis. A polynomial basis of $\mathbb{F}_{2^n}$ is a basis of the form $(1, \omega, \omega^2, \cdots, \omega^{n-1})$, where ω is a root of an irreducible polynomial $f(x)$ of degree n over $\mathbb{F}_2$. In this basis elements of the finite field are represented by polynomials of degree smaller than n and operations are done by means of polynomial arithmetic modulo $f(x)$.

Another representation for finite fields is the normal basis representation. Here a basis of the form $(\alpha, \alpha^2, \cdots, \alpha^{2^{n-1}})$ is used for the finite field $\mathbb{F}_{2^n}$. It is easily verifiable that squaring in this basis can be done using only a circular shift. Multiplication in this basis is more complicated than in the polynomial basis. Further information about finite fields and bases can be found in several books, e.g., McEliece (1987).

2.2.2 Multiplication

Multiplication and inversion are the most resource consuming operations in elliptic curve cryptography. However, although inversion requires more space and time than multiplication it is possible to use a single inversion for the whole scalar multiplication by means of appropriate point representations. It is also imperative to optimize the multiplication algorithms.

Finite field multipliers, depending on the generated bits per clock cycle, can be grouped into the three categories of serial, parallel, and serial-parallel multipliers. The general structure of a finite field multiplier for $\mathbb{F}_{2^n}$, together with the timings of the three groups are shown in Figure 2.2.

We consider only parallel-in multipliers, meaning that the bits of the representations

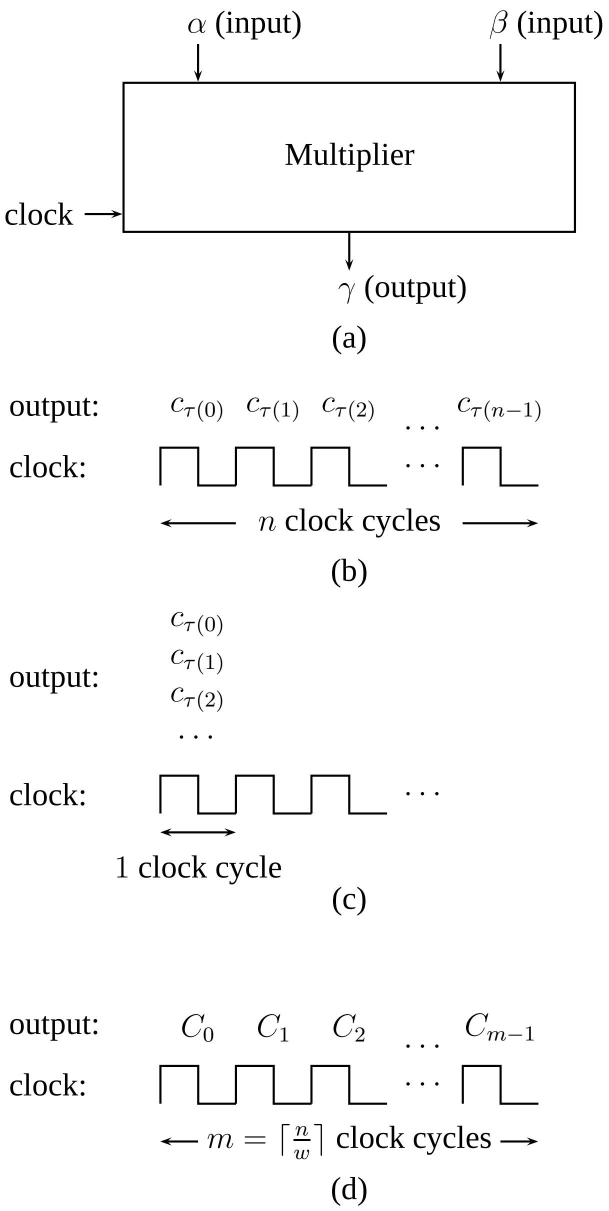


Figure 2.2: (a) The general structure of $\mathbb{F}_{2^n}$ multipliers, together with the timing diagrams of (b) serial, (c) parallel, and (d) serial-parallel multipliers of word-length w. The elements α and β are multiplied to get their product, γ.

of inputs are simultaneously loaded into the multiplier. This requires that each of the input buses be n-bits wide. The clock signal, like other sequential designs, specifies the timing. The rising edge of each clock cycle defines the beginning of one time-interval. The period of the clock signal cannot be arbitrarily short. To see why consider the multiplier block which contains both logic elements and flip-flops. When the inputs of a path, which consists of logic elements only, are applied there is some time needed for its output to be valid and the inputs should remain constant over this time. There is also the settling-time requirement. The settling-time is the time during which the input-pin of a flip-flop must remain stable before the sample-pin of the flip-flop is deactivated. The clock period should not be shorter than the sum of these times. We refer to this sum by the "delay" or the "minimum clock-period". Obviously the multiplication time is the product of the number of clock cycles and this delay.

Figure 2.2-b shows the timing of a serial multiplier. A serial multiplier generates each of the output bits in one clock cycle, hence it requires n clock cycles for a multiplication in $\mathbb{F}_{2^n}$. The sequence of output bits, $c_{\tau(0)}$, $c_{\tau(1)}$, $\cdots$, $c_{\tau(n-1)}$, i.e., the bits of the representation of the product γ can have the same or the reverse ordering as $c_0, c_1, \cdots, c_{n-1}$.

Parallel multipliers, whose timing is shown in Figure 2.2-c, generate all of the output bits in a single clock cycle. The output-bus is in this case n-bits wide. The serial-parallel multipliers fill the gap between the serial and the parallel multipliers. They generate $w > 1$ bits of output in each clock cycle[1]. These sets of w bits are shown as C_0, C_1, $\cdots$, C_{m-1} in Figure 2.2-d. The parameter w is henceforth referred to by "word-length". A serial-parallel multiplier of word-length w performs a multiplication in $\mathbb{F}_{2^n}$ in $\lceil n/w \rceil$ clock cycles.

It should be mentioned that there are other parallel multipliers which require k cycles to compute the result, but in this time other data can be fed to them to be processed. We

[1] Each serial multiplier can also be considered as a special case of serial-parallel with $w = 1$. The reason for the separation of these two concepts in this text is that there are arithmetic methods which are serial but do not possess any direct serial-parallel implementation.

categorize them depending on their application. If they are pipelined multipliers and there are several input values to be fed into these multipliers sequentially we group them as parallel multipliers. The reason is that the multiplication of t values in this case requires $m + t - 1$ cycles. The parameter t becomes insignificant for large values of m and effectively only one clock cycle has been used. If on the other hand no new input can be loaded during the multiplication, either due to the structure of the multiplier or because there are not enough input-data available, we assume the multiplier to be serial-parallel. In all of these cases the multiplication time is the minimum clock-period times the number of clock cycles. Parallel multipliers are generally characterized by large area and delays. They are used for small input lengths. Serial multipliers allow smaller area and shorter delays. They are used when there is only a limited amount of area on the chip.

In this section we discuss only multipliers with low number of bits per clock cycle, i.e., we assume that many clock cycles are required for a single multiplication. Some parallel multipliers will be studied in the next two chapters. The multipliers which we analyze in this section are linear feedback shift register (LFSR) and Massey-Omura (MO) multipliers. These are the two most popular serial-parallel units for polynomial and normal bases respectively. We analyze and compare them in the following three models to reflect different abstraction levels of a circuit (See Bednara *et al.* (2002a) and Bednara *et al.* (2002b)).

- **Theoretical 2-input gate:** This is the most popular model in the literature. It is very well suited to analyze the gate complexity of ASIC or VLSI based hardware modules. But its time analysis results are inaccurate especially in FPGAs, since they do not reflect the delay of buffers used in high fan-out paths or routing elements which are used in FPGAs.

- **FPGA 4-input LUT model:** This is a more practical abstraction of many FPGA based circuits. This model does not only compute the number of 4-input units (like LUTs[2]) but also estimates the propagation delays corresponding to buffers in high fan-out nets. These results can be extracted from the timing analyzer before running the "Place and Route" (par) program. This program is the final part during the synthesization of a circuit for FPGA implementation. When every block of the hardware design is converted to segments which exist on the FPGA and a net-list is generated, this program finds the appropriate positions and connections on the target FPGA and generates a binary configuration file (the bitstream file) which can be downloaded onto the FPGA.

- **FPGA model:** This description of the circuit contains real gate and time complexities of the circuit when implemented on the platform FPGA. Space complexity is computed as the number of used slices and timing complexity as the minimum allowable period for the clock signal across the circuit multiplied by the number of clock cycles required to compute the result. The clock period depends on the propagation delay which contains delays of logic elements, high fan-out buffers, and routing resources. The costs in this model will generally depend on the implemented circuit which will not be unique due to the used nondeterministic place and route algorithms. To achieve more convergent results we set tight timing constraints for "par".

2-input Gate Model

The LFSR multiplier is best known because of its simplicity to perform finite field multiplication in polynomial basis. It generates, in its simplest form, a single bit of output in each clock cycle, but can be easily extended to a serial-parallel multiplier. A schematic diagram of such a multiplier for $\mathbb{F}_{2^n}$ is shown in Figure 2.3. In this figure $m = \lceil \frac{n}{w} \rceil$, where w is the *word-length* or the number of generated bits per clock cycle.

[2] Lookup tables

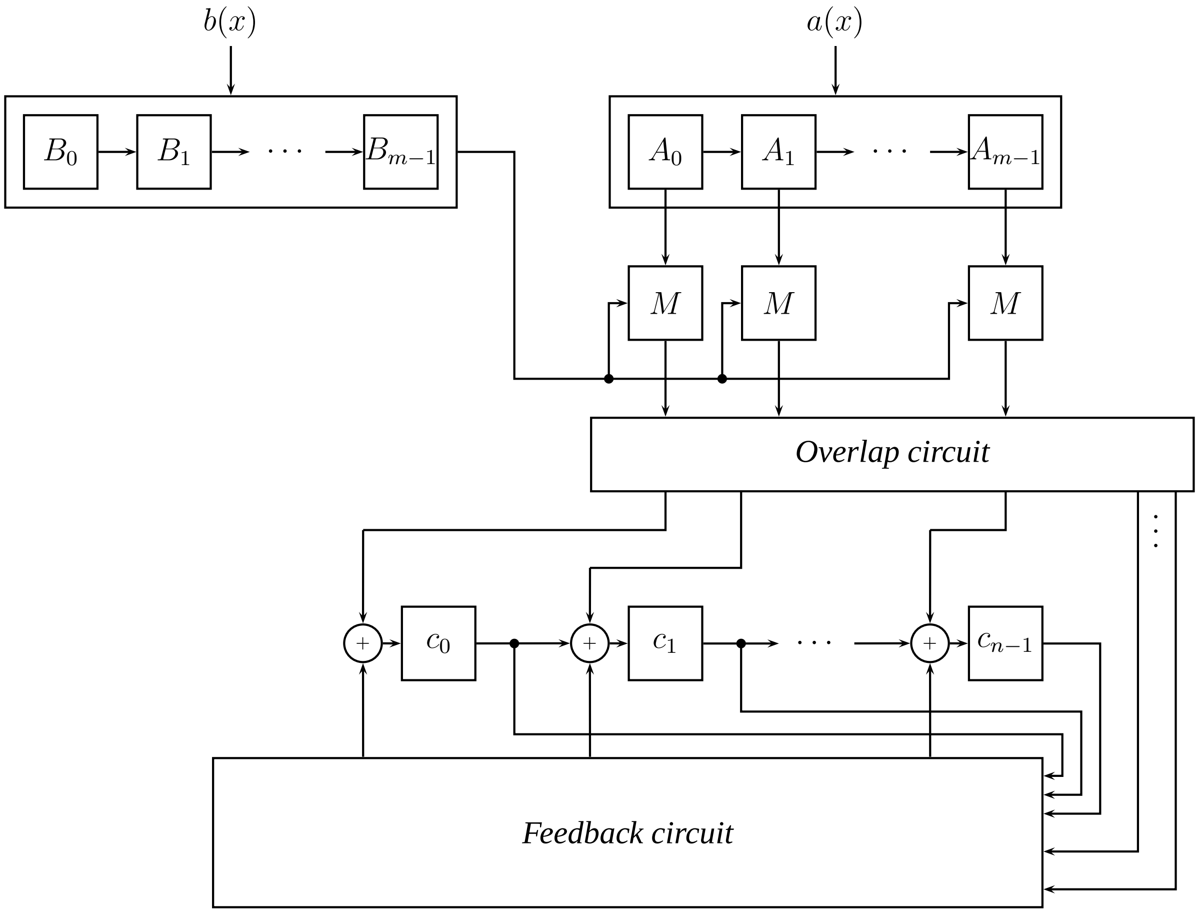


Figure 2.3: Schematic diagram of a serial-parallel LFSR multiplier

At the beginning the polynomials $a(x) = \sum_{i=0}^{n-1} a_i x^i$ and $b(x) = \sum_{i=0}^{n-1} b_i x^i$ are loaded into the word registers A and B to generate $\sum_{j=0}^{m-1} A_j x^{jw}$ and $\sum_{j=0}^{m-1} B_j x^{jw}$ respectively, where each A_j and B_j are polynomials of degree smaller than w. The word multipliers M multiply the highest word of the register B by the words of A. The Overlap circuit adds the coefficients of common powers with each other. In each clock cycle the registers B and C will be shifted to right by w bits, which is equivalent to multiplying by x^w. During shifting C to right, some powers of x will be generated which are greater than or equal to n and should be converted to their representation in the polynomial basis. This will be accomplished by the feedback circuit which hardwires these to smaller powers of x according to the polynomial basis representation of x^i for $n \leq i < n + w$. The product of $a(x)$ and B_k is a polynomial of degree $n + w - 2$ which is again larger than $n - 1$ when $w > 1$. We call the action of converting the produced powers which are greater than $n - 1$ into the polynomial basis $\mathcal{P}$ “Feed forwarding”. This task will also be done using the “Feedback circuit”. Theorem 1 states the space and time complexities of this multiplier.

Theorem 1. *Let $\mathcal{P}$ be a polynomial basis for $\mathbb{F}_{2^n}$ generated by the irreducible polynomial $f(x) \in \mathbb{F}_2[x]$. In an LFSR multiplier of word length w for $\mathcal{P}$ the required number of* AND *gates is mw^2 and the number of* XOR *gates is*

$$(w-1)(mw-1) + n - 1 + H(x^{w+n-1}) + 2\sum_{i=n}^{w+n-2} H(x^i).$$

Here $m = \lceil \frac{n}{w} \rceil$ and $H(x^i)$ is the Hamming weight, or the number of nonzero coefficients, in the representation of x^i in the basis $\mathcal{P}$.

Proof. Each of the m word multipliers require w^2 AND and $(w-1)^2$ XOR gates. The ith word multiplier computes the powers $x^{w(i-1)}$ to $x^{w(i-1)+2w-2}$. Hence, the ith & the $(i+1)$st multipliers have $w - 1$ common coefficients. There are $m - 1$ overlap modules which require in total $(m-1)(w-1)$ XOR gates. Output bits of the overlap circuit can be categorized into two groups, namely the powers smaller than n and the powers which are greater than or equal to n. Adding the first group to the contents of memory cells during shifting in the register C requires $n-1$ XOR gates (the constant coefficient has no neighbor on the left side and requires no addition). But the other group should be computed in the basis $\mathcal{P}$ and added to the register values. It will be done by $\sum_{i=n}^{w+n-2} H(x^i)$ XOR gates. Finally the feedback circuit has to increment the register values by the polynomial basis representation of the high powers of x generated by shift to right. It requires $\sum_{i=n}^{w+n-1} H(x^i)$ XOR gates. Table 2.4 summarizes these results. □

Module		AND gates	XOR gates
Word multipliers		mw^2	$m(w-1)^2$
Overlap circuit		0	$(m-1)(w-1)$
Overlap circuit to register C	$x^0 \cdots x^{n-1}$	0	$n-1$
	$x^n \cdots x^{w+n-2}$	0	$\sum_{i=n}^{w+n-2} H(x^i)$
Feedback module		0	$\sum_{i=n}^{w+n-1} H(x^i)$

Table 2.4: Number of gates in a serial-parallel LFSR multiplier.

The propagation delay depends on the distribution of ones in the polynomial $f(x)$. If representations of no two different powers x^i and x^j for $n \leq i, j < n+w$ have the same nonzero coefficients, the feedback circuit will contribute to an increment of at most two gates. One for the power generated by the shifting and one from the parallel multipliers. For an irreducible polynomial $f(x) = x^n + \sum_{i=1}^{r} x^{n-s_i}$, where s_i is an ascending sequence of positive numbers, this happens if $w < s_1$. For example for the two cases that the irreducible polynomials are trinomials and pentanomials $r = 2, 4$, respectively. The next corollary computes the area and time complexities of the LFSR multiplier for small values of w.

Corollary 2. *Let $\mathcal{P}$ be a polynomial basis for $\mathbb{F}_{2^n}$ generated by the irreducible polynomial $x^n + \sum_{i=1}^{r} x^{n-s_i}$, where $s_i < s_j$ if $i < j$. If the word length w is smaller than s_1 then the area and minimum clock periods of an LFSR multiplier in this basis are given by*

i: $A_{LFSR}(n, \mathcal{P}, w) = mw^2 + (w-1)(mw + 2r - 1) + n - 1 + r,$

ii:

$$D_{LFSR}(n, \mathcal{P}, w) = \begin{cases} T_A + 2T_X & \text{if } w = 1, and \\ T_A + (3 + \lceil \log_2(w) \rceil)T_X & \text{if } w > 1 \end{cases}$$

respectively. Here T_A is the delay of an `AND` *gate, and T_X is the delay of an* `XOR` *gate.*

Proof. The area complexity (case *i) can be computed by setting $H(x^i)$ to r in Theorem 1. To compute the minimum clock period in case ii we observe that each parallel multiplier has a delay of $T_A + \lceil \log_2(w) \rceil T_X$. The overlap circuit, shift register adders, and feedback circuit, according to what already mentioned for the case $w < s_1$, result in a delay of $2T_X$ for $w = 1$ and $3T_X$ if $w > 1$ (there is no overlap circuit if $w = 1$).* □

It is also known that in a finite field $\mathbb{F}_{2^n}$, in which an optimal normal basis of type 2 exists, a Massey-Omura multiplier of word length w requires wn and $w(2n-2)$ gates of types `AND` & `XOR` respectively and has a propagation delay of $T_A + (1 + \lceil \log_2(n-1) \rceil)T_X$ (See Koç & Sunar (1998)).

A comparison of the two multipliers in the 2-input gate model for $\mathbb{F}_{2^{191}}$ is shown in Figure 2.5. Here the computation time, as the product of the number of clock cycles by the minimum clock-period, as a function of required area is plotted. Values are computed for different word lengths w. The polynomial basis $\mathcal{P}$ is generated using the irreducible polynomial $x^{191} + x^9 + 1$ and $\mathbb{F}_{2^{191}}$ contains an optimal normal basis of type 2. As it can be seen the LFSR multiplier is dominant in all practical operating points.

Table 2.6 displays the comparison of the two multipliers in the 4-input LUT and FPGA models[3]. The area in these two models are equal and the minimum clock-periods are shown in the second and third columns for each multiplier respectively. It can be seen

[3] A Massey-Omura multiplier for $w = 96$ does not fit on our FPGA and no delay can be computed.

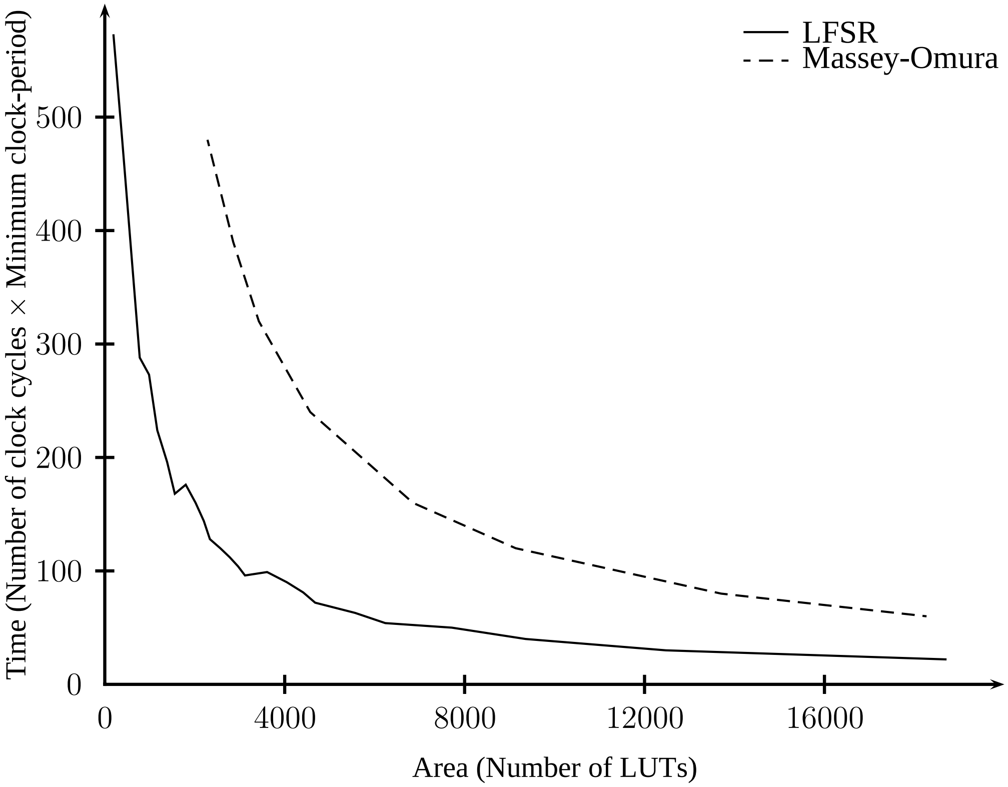


Figure 2.5: Time versus Area comparison of LFSR and Massey-Omura multipliers in $\mathbb{F}_{2^{191}}$

from Table 2.6, that the delay grows faster than estimated when the multiplier gets larger. An effect which is caused by the routing resources.

Bits per	**LFSR**			**Massey-Omura**		
clock	Slice	Delay (ns)	Delay (ns)	Slice	Delay (ns)	Delay (ns)
		(4-input LUT)	(actual)		(4-input LUT)	(actual)
1	288	1.577	3.136	397	7.506	9.847
2	383	2.116	4.295	509	7.506	10.619
4	436	3.194	4.967	730	7.506	12.670
8	762	3.733	6.278	1172	7.506	15.666
16	1513	4.811	11.554	2052	7.506	18.403
32	2558	5.889	15.423	3814	7.506	16.568
48	3642	8.584	21.745	5584	7.506	26.720
64	4712	7.506	22.419	7347	7.506	26.886
96	6837	7.506	27.846	10847	–	–

Table 2.6: Comparing the LFSR and Massey-Omura multipliers in $\mathbb{F}_{2^{191}}$ implemented on a XCV2000e FPGA. Delays are the minimum clock period in nano-seconds for the 4-input LUT model and the actual FPGA implementations respectively.

2.2.3 Squaring

Another important operation in elliptic curve cryptography is the squaring. It can be done more efficiently than multiplication. For comparison we consider again two different cases of normal and polynomial bases.

Normal Bases

Squaring an element which is represented in normal basis requires only a cyclic shift of the corresponding vector. We assume the space and time complexities of this operation to

2-input gate		**4-input LUT**		**FPGA model**	
Space	Delay	Space	Delay	Space	Delay
95	$2T_X$	91	$6.477ns$	91	$8.012ns$

Table 2.7: Space and time complexities of squaring in $\mathbb{F}_{2^{191}}$ using three different models.

be 0.

Polynomial Bases

Computing the square of a polynomial over $\mathbb{F}_2$ can be easily done by inserting zeros between each two adjacent coefficients. The resulting polynomial should then be reduced modulo the irreducible polynomial characterizing the basis. Some upper bounds for the space and time complexities are reported in Wu (2000). If the irreducible polynomial is of the form $f(x) = x^n + x^k + 1$ and $k < \frac{n}{2}$, then reducing a general polynomial of degree $2n-2$ modulo $f(x)$ can be done using a circuit with at most $2(n-1)$ XOR gates. The depth of the circuit would be at most $2T_X$. Figure 2.8 shows the circuit to perform squaring in $\mathbb{F}_{2^{191}}$. In this figure the circles in the ith column show the input coefficients which must be added to compute the ith output-bit. For example the circles in the gray box show that the coefficient of x in the resulting polynomial is the sum of a_{96} (for x^{192}) and a_{187} (for x^{374}). Here the circles in the first row are the low-order coefficients a_0 to a_{95} of the original polynomial corresponding with the powers 1 to x^{190}.

This kind of squarer is especially attractive for FPGA based circuits where the structure of circuits can be modified in each design depending on the selected finite field. For the case of $\mathbb{F}_{2^{191}}$ we have used the trinomial $x^{191} + x^9 + 1$ to represent the finite field. Results in three models are shown in Table 2.7.

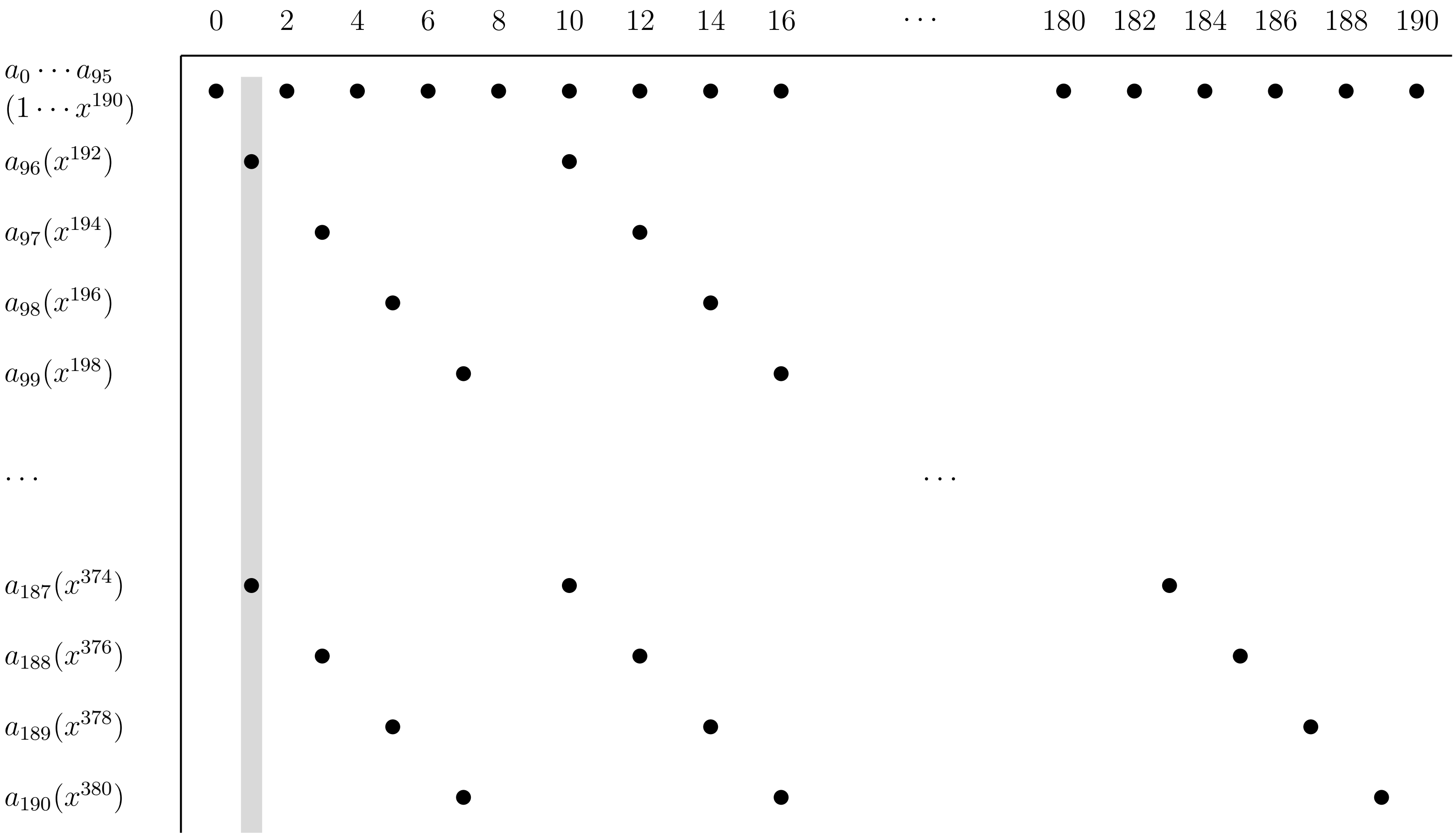


Figure 2.8: The Distribution of additions for squaring in $\mathbb{F}_{2^{191}}$ in the polynomial basis generated by $x^{191} + x^9 + 1$

2.2.4 Inversion

There are generally two different methods for inversion in finite fields, namely the Euclidean algorithm and Fermat's theorem. Classical Euclidean algorithm requires one polynomial division and two multiplications in each stage. There are at most n stages to compute the inverses in $\mathbb{F}_{2^n}$. This fact makes it inefficient for hardware implementations. Instead the binary Euclidean method is often used where only addition of polynomials is required (see Guo & Wang (1998), Takagi (1998), and Shantz (2001)). The binary Euclidean algorithm requires in the worst case $2n$ clocks which cannot be reduced using more space. We use here the second method which is based on the fact that:

$$x^{2^n-2} = x^{-1} \tag{2.1}$$

for any element $x \in \mathbb{F}_{2^n}^{\times}$. To compute the $(2^n - 2)$th power of an element we use the method by Asano *et al.* (1989) and von zur Gathen & Nöcker (2003). Application of this method to $\mathbb{F}_{2^{191}}$ can be summarized as represented in Figure 2.9. As can be seen an inversion requires 10 multiplications and 190 squarings independent of the finite field representation.

2.3 Point Addition and Doubling

Point arithmetic is another building block for multiplication in elliptic curves. As it is mentioned in Chapter 1, points on an elliptic curve together with the point at infinity $\mathcal{O}$ form an abelian group. There have been different proposals for point representations. Each of them has its own advantages and drawbacks and some of them are suitable for special implementation platforms. For a detailed survey of different representations in prime finite fields see Cohen *et al.* (1998); López & Dahab (1999a) give a survey for binary finite fields. We review such representations and their resource consumptions for parallel implementations. Most of these works are already cited in Bednara *et al.* (2002a) and Bednara *et al.* (2002b). In our analysis we count only the number of inversions and

```
0:   $y_1 \leftarrow x$                                   $\{x^{2^1-1}\}$
1:   $y_2 \leftarrow y_1^2 \cdot y_1$                     $\{x^{2^2-1}\}$
2:   $y_3 \leftarrow y_2^2 \cdot y_1$                     $\{x^{2^3-1}\}$
3:   $y_5 \leftarrow y_3^{2^2} \cdot y_2$                 $\{x^{2^5-1}\}$
4:   $y_{10} \leftarrow y_5^{2^5} \cdot y_5$              $\{x^{2^{10}-1}\}$
5:   $y_{20} \leftarrow y_{10}^{2^{10}} \cdot y_{10}$     $\{x^{2^{20}-1}\}$
6:   $y_{40} \leftarrow y_{20}^{2^{20}} \cdot y_{20}$     $\{x^{2^{40}-1}\}$
7:   $y_{80} \leftarrow y_{40}^{2^{40}} \cdot y_{40}$     $\{x^{2^{80}-1}\}$
8:   $y_{85} \leftarrow y_{80}^{2^5} \cdot y_5$           $\{x^{2^{85}-1}\}$
9:   $y_{95} \leftarrow y_{85}^{2^{10}} \cdot y_{10}$     $\{x^{2^{95}-1}\}$
10:  $y_{190} \leftarrow y_{95}^{2^{95}} \cdot y_{95}$    $\{x^{2^{190}-1}\}$
11:  Output $\leftarrow y_{190}^2$                        $\{x^{2^{191}-2}\}$
```

Figure 2.9: Sequence of multiplications and squarings for inversion in $F_{2^{191}}$

multiplications because of their higher costs compared to addition and squaring in FPGA designs.

Parallel implementations are interesting both from hardware and software point of views. It is shown in Section 2.2.2 that multiplication costs grow faster than linear when the word length is increased. In hardware designs, this suggests breaking up large multipliers into as many parallel multipliers[4] as possible. Efficient arithmetic with processors which contain several ALUs like C6000 DSP series requires parallel algorithms to be developed. This would be also advantageous for better use of pipeline stages in RISC processors (see Hennesy & Patterson (2003)). An analysis of possible parallelism for some special types of finite fields has been already described by Smart (2001). We consider here again only fields of characteristic 2 since they are more suitable for hardware implementations.

[4] We assume efficient communication inside the FPGA.

2.3.1 Simple Representations

Possibly the most straightforward representation of points on elliptic curves is the affine representation from which other representations can be derived. Here every point is specified using two coordinates x and y. We consider the general equation of a non-supersingular elliptic curve over a field of characteristic 2 according to Blake *et al.* (1999), namely:

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

Two different points $\mathcal{Q}_1 = (x_1, y_1)$ and $\mathcal{Q}_2 = (x_2, y_2)$ can be added to result in a third point $\mathcal{Q}_3 = (x_3, y_3)$ using the following formula if $x_1 + x_2 \neq 0$:

$$\begin{aligned} \lambda &= \frac{y_1 + y_2}{x_1 + x_2}, \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a, \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1. \end{aligned} \tag{2.2}$$

The sum of two different points if the sum of their x-coordinates is 0 is the point at infinity, or $\mathcal{O}$.

The point $\mathcal{Q}_4 = (x_4, y_4) = 2\mathcal{Q}_1$, if $x_1 \neq 0$, can be computed by:

$$\begin{aligned} \lambda &= \frac{y_1}{x_1} + x_1, \\ x_4 &= \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2}, \\ y_4 &= (x_1 + x_4)\lambda + x_4 + y_1. \end{aligned} \tag{2.3}$$

If $x_1 = 0$ then $\mathcal{Q}_4$ will be the point at infinity.

As can be seen, each of the addition or doubling operations requires 1 inversion, 2 multiplications, and 1 squaring or 1 division, 1 multiplication, and 1 squaring. We denote this cost by $1I + 2M$ since the costs of addition and squaring in the field are negligible. The second multiplication in each of these relations depends on the first one and they cannot be performed in parallel.

Representation	Mappings		Addition		Doubling	
	x	y	costs	depth	costs	depth
Jacobian ($\mathfrak{J}$)	X/Z^2	Y/Z^3	$16M$	$4M$	$5M$	$2M$
López/Dahab ($\mathfrak{L}$)	X/Z	Y/Z^2	$13M$	$4M$	$4M$	$2M$

Table 2.10: Some elliptic curve point representations with their corresponding costs.

A possibility to avoid inversion in each point addition and doubling is to use a projective point representation. In this case x and y from the affine representation are substituted by X/Z^m and Y/Z^n for some specific values of m and n which result in different projective representations. All points $(\alpha X, \alpha Y, \alpha Z)$ for $\alpha \neq 0$ belong to the same equivalence class which is represented by $(X : Y : Z)$. Setting the Z-coordinate equal to 1 results in the same X and Y coordinates as the corresponding affine representation. The point at infinity will be the equivalence class $(0 : 1 : 0)$. Here we consider the most popular Jacobian and the most efficient López-Dahab representations. Table 2.10 summarizes these representations, their costs, and the length of the longest computation path when a parallel implementation is used. It should be mentioned that the Jacobian addition can be done with fewer multiplications, but that implementation has larger depth and is not efficient for parallel implementation. Data dependency diagrams for some of these representations are shown in Figures 2.11, 2.12, 2.13, and 2.14.

2.3.2 Mixed Representations

Mixed representations for elliptic curves have been independently published by Cohen *et al.* (1998) and López & Dahab (1999b) for prime and binary finite fields respectively. Indeed the work by Cohen *et al.* (1998) is more general and can be applied to simple double-and-add and addition-subtraction chains as well as other sophisticated methods like Brauer methods (see Brauer (1939)). Because of memory limitations in FPGA circuits we do not consider windowing methods and discuss parallel implementation of the method by López & Dahab (1999b) only.

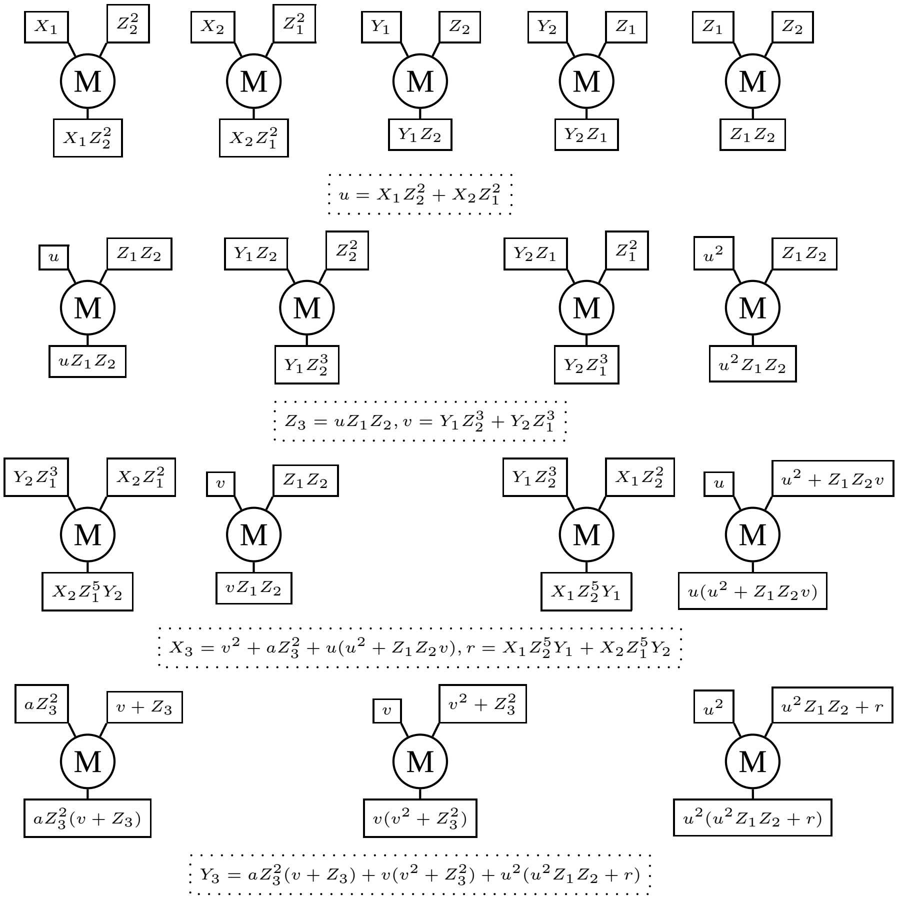


Figure 2.11: Data dependency of parallel implementation of point addition in Jacobian representation

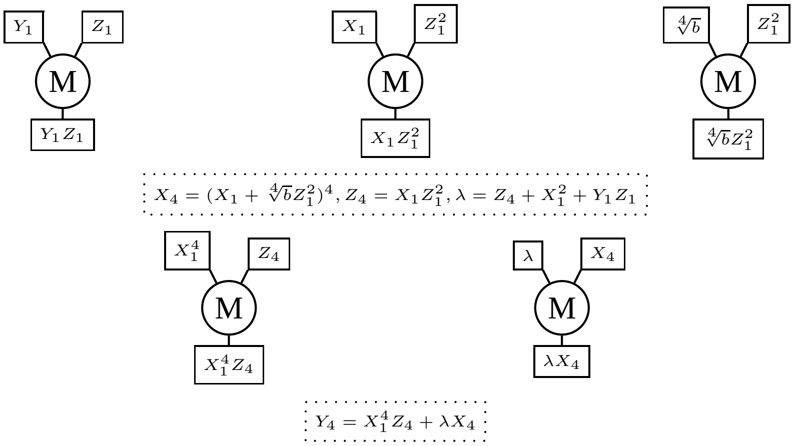


Figure 2.12: Data dependency of parallel implementation of point doubling in Jacobin representation

Representation	Addition		Doubling	
	costs	depth	costs	depth
Mixed Jacobian ($\mathcal{J}$)	$12M$	$4M$	$5M$	$2M$
Mixed López/Dahab ($\mathcal{L}$)	$10M$	$4M$	$4M$	$2M$

Table 2.15: Addition and doubling costs for mixed representation arithmetic over elliptic curves.

The mixed López-Dahab and mixed Jacobian addition methods are other kinds of projective additions in which the Z coordinate of one of the points is set to 1. In this way some multiplications and squarings can be saved resulting in new addition formulas which are shown in Figures 2.16 and 2.17. The new costs are shown in Table 2.15.

2.4 Scalar Multiplication

Scalar multiplication or the task of computing $m\mathcal{Q}$, for a given integer m and a point $\mathcal{Q}$, consists of many additions, doublings, and possibly negations which must be combined together. The selected method to perform these simpler operations depends on several

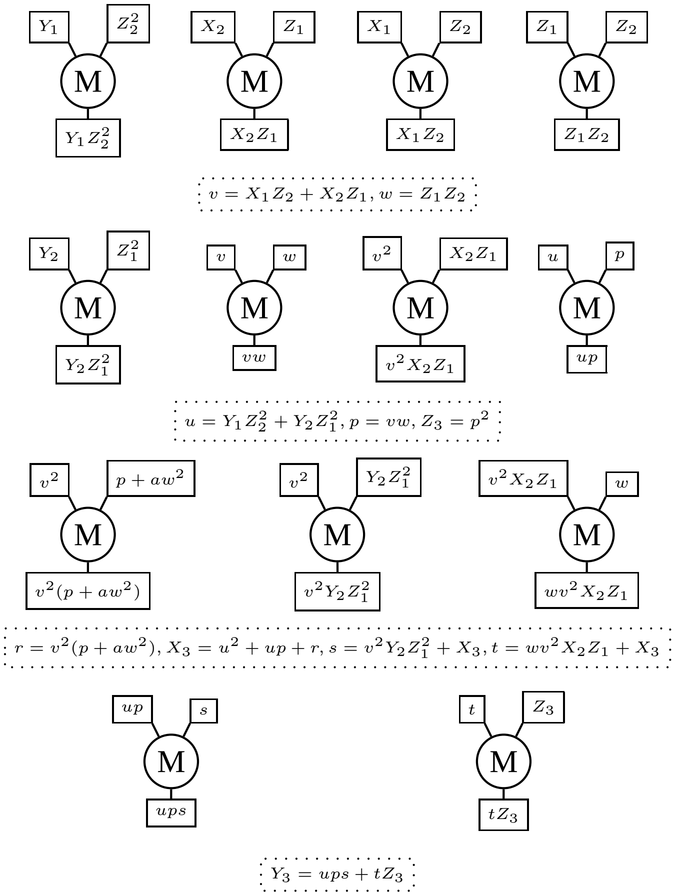


Figure 2.13: Data dependency of parallel implementation of point addition in López/Dahab representation

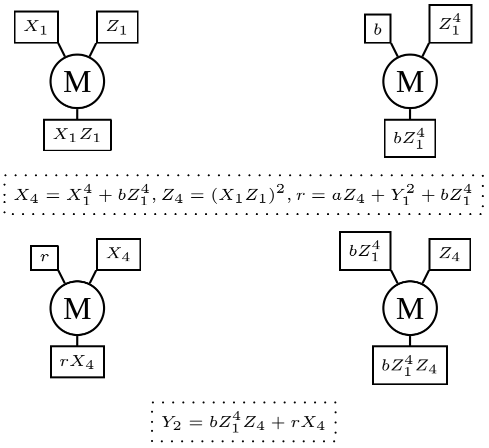


Figure 2.14: Data dependency of parallel implementation of point doubling in López/Dahab representation

parameters like the point arithmetic costs and the amount of available memory (for precomputation and intermediate results). FPGAs have only limited amount of memory, and these memory blocks (see Block SelectRAM in Section 1.3) are distributed across the FPGA. Accessing these memory cells is one of the slowest operations inside FPGAs. Hence, we limit our study to methods which do not require precomputations.

There are three methods which can be applied here: The “double and add” and the “addition-subtraction chains” methods which are thoroughly investigated in the literature (see Knuth (1998), Morain & Olivos (1990), and Otto (2001)), and the Montgomery method which is developed by Montgomery (1987). López & Dahab (1999a) have used the closed form formulas of point addition and doubling in affine representation to apply the method to fields of characteristic 2. The “double and add” method requires on average n doublings and $n/2$ additions for multiplication of $\mathcal{Q}$ by a n bit random number. Addition-subtraction chains which are introduced by Morain & Olivos (1990) insert subtractions into addition chains. These structures decrease the number of operations to n doublings and $n/3$ additions on average. The Montgomery method requires exactly n doublings and n additions for the complete multiplication, which is more than other methods, but in each of these operations, only X and Z coordinates have to be com-

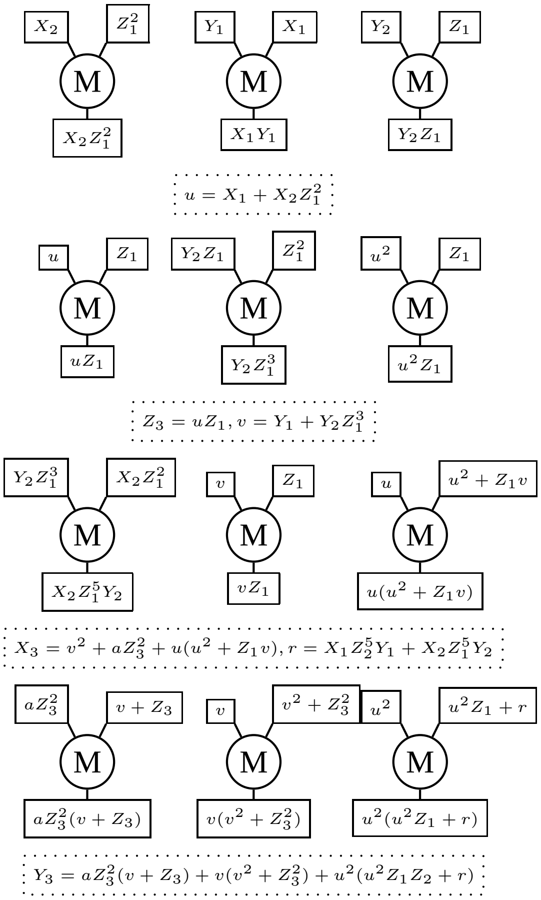


Figure 2.16: Data dependency of parallel implementation of mixed mode point addition in Jacobian representation

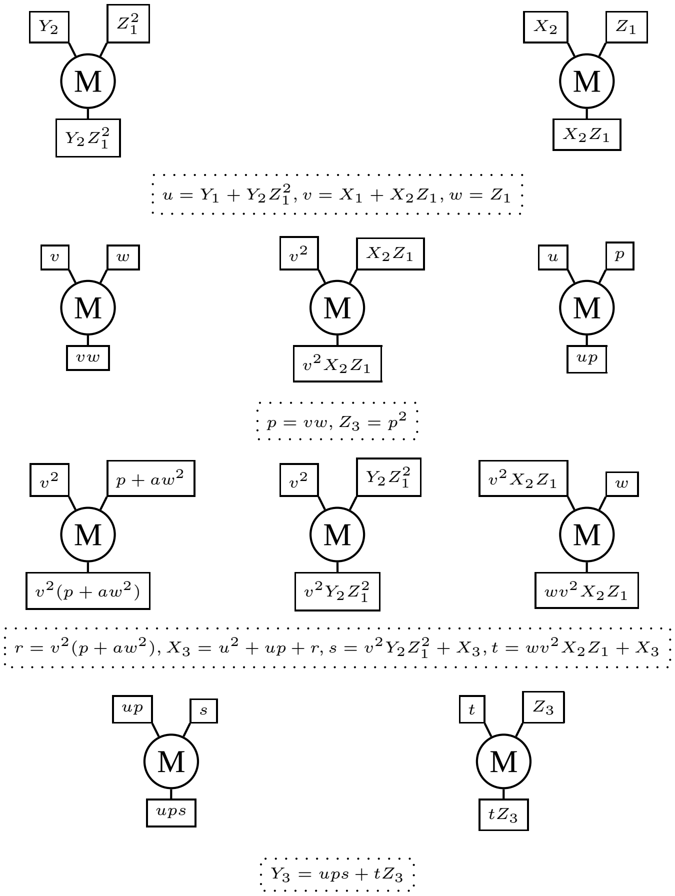


Figure 2.17: Data dependency of parallel implementation of mixed mode point addition in López/Dahab representation

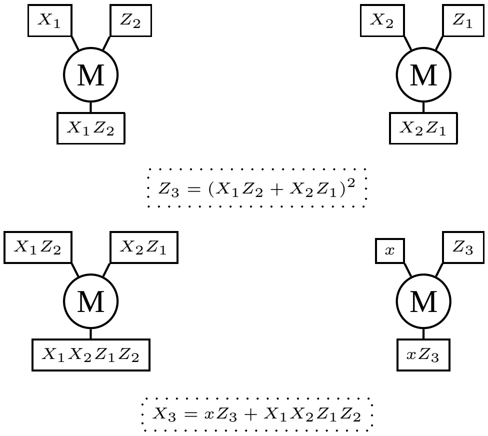


Figure 2.18: Parallel implementation of point addition in the Montgomery method, in which x is the x-coordinate of the difference of the two points which are added together.

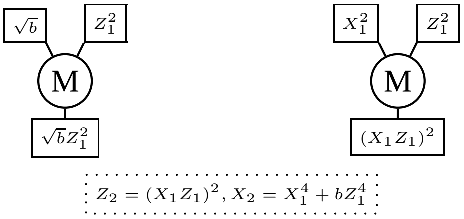


Figure 2.19: Parallel implementation of point doubling in the Montgomery method

puted. The computation of the Y coordinate is postponed to the last stage. Figure 2.4 compares the average number of required multiplications in the best method for each of these representations. In part (a) of this figure the average number of multiplications are compared whereas in part (b) the average number of cascaded multiplications in parallel implementations is shown.

Data dependency diagrams for addition and doubling in the Montgomery method are shown in Figures 2.18 and 2.19, respectively. In Figure 2.18 two points $\mathcal{R}$ and $\mathcal{S}$ are added such that $\mathcal{R} - \mathcal{S} = \mathcal{Q}$ and x the x-coordinate of $\mathcal{Q}$ is already known.

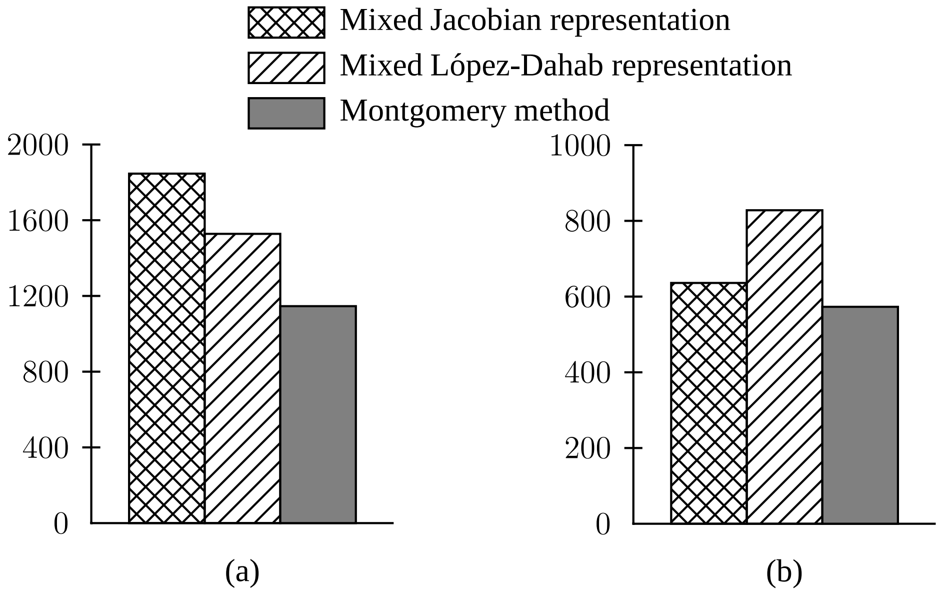


Figure 2.20: Average numbers of (a) total and (b) cascaded finite field multiplications to multiply a point on an elliptic curve by a random scalar m, $(2^{190} < m < 2^{191})$, when addition-subtraction chains are used for the parallel versions of Jacobian and López-Dahab methods.

2.5 FPGA-Based Co-Processor

As we have seen in Section 2.4 if precomputation is avoided the Montgomery method results in the best performance. Based on this observation we have implemented an ECCo (Elliptic Curve Co-processor) (see Bednara *et al.* (2002a) and Bednara *et al.* (2002b)). There are several such implementations, see Gao *et al.* (1999), Orlando & Paar (2000), and Goodman & Chandrakasan (2001) for example. The distinguishing factor in our work is the deployment of parallel units. Because of different platform FPGAs, a direct comparison of these implementations is not possible. But we have shown in Section 2.2.2 that parallel small multipliers (as long as parallelism is possible) would result in a better performance than larger multipliers.

2.5.1 Data-path Architecture

The generic data-path architecture for the co-processor is shown in Figure 2.21. It is based on the implementation of Daldrup (2002) which uses the mixed representation. Flexibility due to finite field extension is achieved by using "generic" parameters. Modules which could not be parameterized using VHDL structures are produced by means of code generators which are written in C++.

Modularity of the structure makes it flexible to meet various performance/area constraints which we describe in subsequent sections. This structure also lets prototyping of other point representations by modifying the state machine. Particular modules of the system are described in the following paragraphs:

- **Dual port RAM**

  This memory stores input and output data together with the intermediate results of computations. This module is implemented using the Block SelectRAM modules which are briefly explained in Section 1.3. The width of its I/O buses is equal to the extension degree n of the selected finite field $\mathbb{F}_{2^n}$ since it should store and load polynomials of degrees smaller than n. One of its ports will be written by

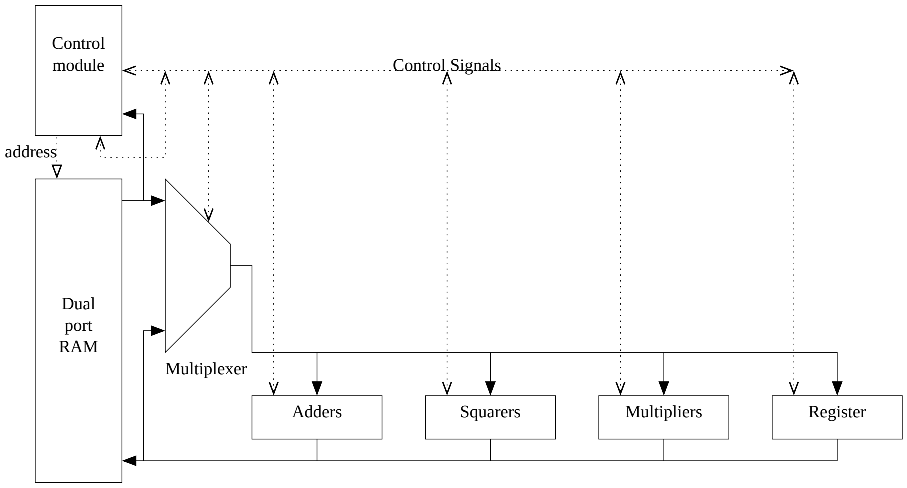


Figure 2.21: Data-path structure.

a host processor interface module and the other one will be connected through a multiplexer to the arithmetic modules.

- **Multiplexer**

  There are three groups of data buses in the processor, mainly:

  - input data to the dual port RAM which are also the outputs of arithmetic units,
  - output data from the dual port RAM,
  - and input data to the arithmetic units.

  Saving and loading data to and from the dual port RAM is always time consuming. It requires at least two clock cycles and it is sometimes more efficient to load an arithmetic unit directly from another or the same module. The multiplexer decides which of the first two data lines should be used to load each arithmetic module.

- **Adders**

  This module consists of two adders, each of them having two input buffers and one output buffer. Each input buffer requires one clock cycle to be loaded. Addition which is only a bitwise XOR combination will be done in one clock cycle. The code uses “generic” parameters.

- **Squarers**

  There are two squarers which can be used in parallel. Their input-output structures are like those of adders. They are generated using a code generator for each field extension.

- **Multipliers**

  As we have already seen at most two multipliers can be deployed at the same time during the Montgomery algorithm. The multipliers are the most time and area consuming elements in our design. They are LFSR multipliers which are generated using a code generator. They are flexible both with respect to polynomial length and parallelism degree. So if there is more space on the platform FPGA, the word-length of the multipliers can be increased. But it should be taken into account that this structure uses extra clock cycles to load and save from and into register files and is effective as long as the multipliers require several clock cycles.

- **Control module**

  This is probably the most complicated module in our ECCo. It controls the overall point multiplication and consists of several other submodules. So we devote a complete section to it.

2.5.2 Control Module

This part is responsible for performing the Montgomery multiplication algorithm. The required sequence of point additions and doublings of this algorithm is shown in Algorithm 3 in which $k = \lceil \log_2 n \rceil$. The point additions and doublings can be in any representations and the y-coordinate needs to be computed only in the last stage. We use additions and doublings as stated in Figures 2.18 and 2.19.

This module consists of a state machine, performing Algorithm 3, which communicates with several other submodules as shown in Figure 2.22. These different submodules are described as follows:

Algorithm 3 The Montgomery point multiplication algorithm expressed in point level.

Input: An elliptic curve with a fixed point $\mathcal{Q}$ on it, together with the binary representation of the scalar multiplier m as $(m_{k-1}m_{k-2}\ldots m_1m_0)_2$.

Output: $m\mathcal{Q}$

```
$\mathcal{Q}_1 \leftarrow \mathcal{Q}$, $\mathcal{Q}_2 \leftarrow 2\mathcal{Q}$
for $i$ from $k-2$ downto $0$ do
    if $m_i = 1$ then
        $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 + \mathcal{Q}_2$, $\mathcal{Q}_2 \leftarrow 2\mathcal{Q}_2$
    else
        $\mathcal{Q}_2 \leftarrow \mathcal{Q}_1 + \mathcal{Q}_2$, $\mathcal{Q}_1 \leftarrow 2\mathcal{Q}_1$
    end if
end for
```

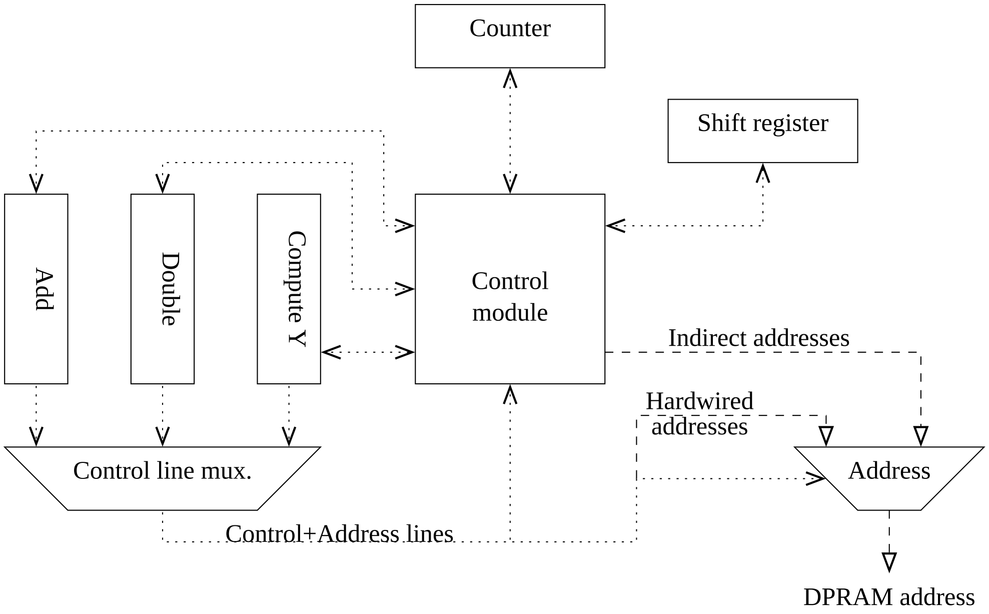


Figure 2.22: The structure of the control module in ECCo.

- **Counter**

  The counter in the control module takes care of the number of iterations in Algorithm 3 to be exactly $\lceil \log_2 n \rceil - 1$ when n is the order of the group of points on the elliptic curve.

- **Shift register**

  This register will be directly loaded from the dual port RAM and contains the multiplier m. By each repetition of Algorithm 3 this register will be shifted to right. The LSB of this register is the decision criterion for the control module state machine.

- **Control module state machine**

  This state machine controls the overall operation of the processor. It starts other modules, gives the control to them, and waits for their terminations.

- **Add, Double, and Compute Y**

  These state machines perform the operations addition, doubling, and computing the y-coordinate. The latter will be activated only once during the total point multiplication. Each of these operations will be started with the command of the control module state machine, which at the same time gives the control of all of the processor elements to these modules. After finishing, they activate a signal in the main state machine which takes their control back by changing the addresses of the control line multiplexer.

- **Control line multiplexer**

  There is a single control bus in the processor which consists of address lines for the dual port RAM, commands to arithmetic units, and their ready status signals. The control module state machine can change the master of this bus by activating the corresponding address in this multiplexer.

- **Address multiplexer**
  As stated above there is a single control line in the processor. The point additions and doublings in Algorithm 3 consist of the same operations which are performed on different variables. Results should also be written back to different addresses. This is done during an indirect addressing process. The control module state machine puts the addresses of the arguments and return values on the indirect address line inputs of the multiplexer. The module which controls the processor can select these addresses by activating the indirect address line on this multiplexer.

2.6 Benchmarks

Using the above structure on a XCV2000e FPGA we have performed a complete point multiplication over $\mathbb{F}_{2^{191}}$ in 0.18 ms using clock frequency of 66 MHz. Our finite field multipliers generate 64 bits in one clock cycle. Since the performance of hardware implementations depends on the amount of used area we can compare our results to Lutz & Hasan (2004) only, which is on the same FPGA and the same clock frequency. Their implementation, which is optimized for Koblitz curves, requires 0.238 ms for a point multiplication on a generic curve over $\mathbb{F}_{2^{163}}$. As a measure of comparison, one of the known running times for point multiplication in software is given by Hankerson *et al.* (2000), where a point multiplication in $\mathbb{F}_{2^{163}}$ is done in 3.24 ms. Assuming a cubic growth factor for the point multiplication (quadratic for finite field multiplications times linear for the size of the scalar m) these hardware and software times can be extrapolated to 0.38 and 5.21 ms over our field $\mathbb{F}_{2^{191}}$, respectively. The above hardware implementation results are summarized in Table 2.23.

Implementation	Multiplication time
Lutz & Hasan (2004) for $\mathbb{F}_{2^{163}}$	0.238 ms
Extrapolating the results of Lutz & Hasan (2004) to $\mathbb{F}_{2^{191}}$	0.38 ms
Our results for $\mathbb{F}_{2^{191}}$	0.18 ms

Table 2.23: Comparisons of different elliptic curve scalar multiplication times on a XCV2000e FPGA with a clock frequency of 66 MHz. To extrapolate the multiplication time of Lutz & Hasan (2004) for $\mathbb{F}_{2^{191}}$ we have used a cubic growth factor.

2.7 Conclusion

This chapter presents the stages of the design and implementation of an FPGA-based co-processor for elliptic curve cryptography. It is shown how optimizations in different levels of finite field arithmetic, point addition and doubling, and scalar multiplication can be combined to achieve a high performance co-processor. Here the scalar multiplication refers to the sequence of additions and doublings which compute an integer multiplier of a point. Finally the data-path of the designed co-processor together with the benchmarks, when implemented on a XCV2000e FPGA are presented and compared with a published result on a similar FPGA. The topics of this chapter are:

- Polynomial and normal bases as two popular finite field representations are compared. This comparison is made for the costs of arithmetic in finite fields for the special case of elliptic curve cryptography using generic curves. It is shown that, especially when serial-parallel multipliers are used, polynomial bases are always better than normal bases.

- Several point representations and their effect on the efficiency of point addition and doubling are compared. The mixed representations of points are discussed.

- The double-and-add, addition-subtraction chains, and the Montgomery method for point multiplication are compared. It is shown that, the Montgomery method requires fewer operations than the other two methods, since it does not compute the y-coordinate at each iteration.

- The structure of an FPGA-based co-processor using the Montgomery method together with the benchmarks, when implemented on a XCV2000e FPGA, are presented.

Chapter 3

Sub-quadratic Multiplication in Hardware

3.1 Introduction

As mentioned in Chapter 2, arithmetic and in particular multiplication in finite fields are central algorithmic tasks in cryptography. The multiplication in our case, in finite fields of characteristic 2, can be achieved by multiplying two polynomials over $\mathbb{F}_2$ followed by a reduction modulo the irreducible polynomial defining the field extension. This reduction can be done again using multiplications or very small circuits.

Classical methods for multiplying two n-bit polynomials require $O(n^2)$ bit operations. The Karatsuba algorithm reduces this to $O(n^{\log_2 3})$ (see Karatsuba & Ofman (1963)). The Fast Fourier Transformations (FFT) with a cost of $O(n \log n \operatorname{loglog} n)$ and the Cantor multiplier with a cost of $O(n(\log n)^2(\operatorname{loglog} n)^3)$ are efficient for high extension degrees and therefore are not studied here for applications to cryptography (see Cantor (1989) and von zur Gathen & Gerhard (1996)).

In this chapter hardware implementations of the Karatsuba method and its variants are studied. Even the Karatsuba method which has the lowest crossover point with the classical algorithm is asymptotically good and thus efficient for large degrees. Sophisticated implementation strategies decrease the crossover point between different algorithms and make them efficient for practical applications.

Efficient software implementations of the Karatsuba multipliers using general purpose processors have been discussed thoroughly in the literature (see Paar (1994), Bailey & Paar (1998), Koç & Erdem (2002), Hankerson *et al.* (2003), Chapter 2, and von zur Gathen & Gerhard (2003), Chapter 8). Hardware implementations to the contrary have attracted less attention. Jung *et al.* (2002) and Weimerskirch & Paar (2003) suggest to use algorithms with $O(n^2)$ operations to multiply polynomials which contain a prime number of bits. Their proposed number of bit operations is by a constant factor smaller than the classical method but asymptotically larger than those for the Karatsuba method. In Grabbe *et al.* (2003a) we have proposed a hybrid implementation of the Karatsuba method which reduces the latency by pipelining and by mixing sequential and combinational circuits.

The goal of this chapter is to decrease the resource usage of polynomial multipliers by means of both known algorithmic and platform dependent methods. This is achieved by computing the best choice of hybrid multiplication algorithms which multiply polynomials with at most 8192 bits. This choice is restricted to six recursive methods, namely: classical, Karatsuba, a variant of Karatsuba for quadratic polynomials, and three methods of Montgomery (2005) for polynomials of degrees 4, 5, and 6, respectively. The “best” refers to minimizing the area measure. This is an algorithmic and machine independent optimization. The 240-bit multiplier of Grabbe *et al.* (2003a) is re-used here, which was implemented on a XC2V6000-4FF1517-4 FPGA, to illustrate a second type of optimization, which is machine-dependent. The goal is a 240-bit multiplier with small area-time cost. A single 30-bit multiplier is put on the used FPGA and three Karatsuba steps are applied to get from $240 = 2^3 \cdot 30$ to 30 bits. This requires judicious application of multiplexer and adder circuitry, but the major computational cost still resides in the 30-bit multiplier. Twenty seven (3^3) small multiplications are required for one 240-bit product and these inputs are fed into the single small multiplier in a pipelined fashion. This has the pleasant effect of keeping the total delay small and the area reduced, with correspondingly small propagation delays. This 240-bit multiplier covers in particular the 233-bit polynomials proposed by NIST for elliptic curve cryptography (FIPS PUB 186-2 (2000)).

Substantial parts of this chapter have been published in Grabbe *et al.* (2003a), Grabbe *et al.* (2003b), von zur Gathen & Shokrollahi (2005), and von zur Gathen & Shokrollahi (2006).

The structure of this chapter is as follows. First the Karatsuba method and its cost are studied in Section 3.2. Section 3.3 is devoted to the optimized hybrid Karatsuba implementations. Section 3.4 shows how a hybrid structure and pipelining can improve resource usage in the circuit of Grabbe *et al.* (2003a). Section 3.5 analyzes the effect of the number of recursion levels on the performance. Section 3.6 briefly describes the structure of our developed code generator for the used combinational pipelined multiplier, and finally Section 3.7 concludes this chapter.

3.2 The Karatsuba Algorithm

The three coefficients of the product $(a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + (a_1b_0 + a_0b_1)x + a_0b_0$ are “classically” computed with 4 multiplications and 1 addition from the four input coefficients a_1, a_0, b_1, and b_0. The following formula uses only 3 multiplications and 4 additions:

$$(a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + ((a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0)x + a_0b_0. \quad (3.1)$$

We call this the 2-*segment Karatsuba method or* K_2. Setting $m = \lceil n/2 \rceil$, two n-bit polynomials (thus of degrees less than n) can be rewritten and multiplied using the formula:

$$(f_1x^m + f_0)(g_1x^m + g_0) = h_2x^{2m} + h_1x^m + h_0,$$

where f_0, f_1, g_0, and g_1 are m-bit polynomials respectively. The polynomials h_0, h_1, and h_2 are computed by applying the Karatsuba algorithm to the polynomials f_0, f_1, g_0, and g_1 as single coefficients and adding coefficients of common powers of x together. This method can be applied recursively. The circuit to perform a single stage is shown in Figure 3.1.

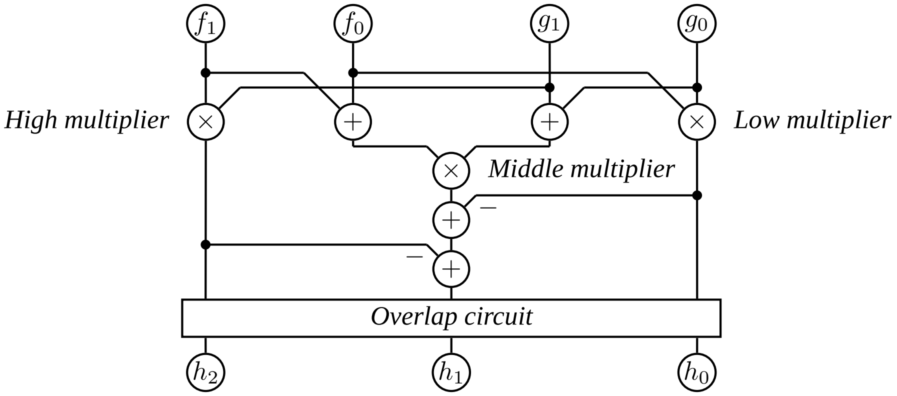


Figure 3.1: The circuit to perform one level of the Karatsuba multiplication

The “overlap circuit” adds common powers of x in the three generated products. For example if $n = 8$, then the input polynomials have degree at most 7, each of the polynomials f_0, f_1, g_0, and g_1 is 4 bits long and thus of degree at most 3, and their products will be of degree at most 6. The effect of the overlap module in this case is represented in Figure 3.2, where coefficients to be added together are shown in the same columns.

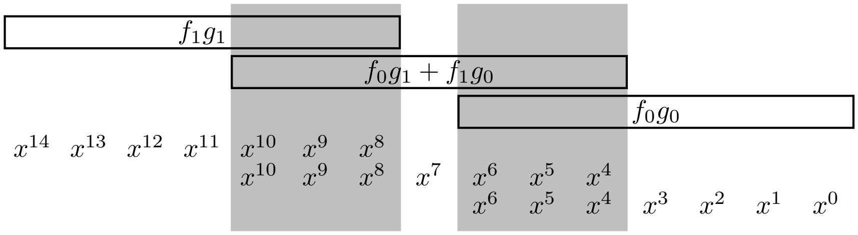

f_1g_1
$f_0g_1 + f_1g_0$
f_0g_0
x^{14} x^{13} x^{12} x^{11} x^{10} x^9 x^8
x^{10} x^9 x^8 x^7 x^6 x^5 x^4
x^6 x^5 x^4 x^3 x^2 x^1 x^0

Figure 3.2: The overlap circuit for the 8-bit Karatsuba multiplier

Figures 3.1 and 3.2 show that we need three multiplication calls at size $m = \lceil n/2 \rceil$ and some adders: 2 for input, 2 for output, and 2 for the overlap module of lengths m, $2m - 1$, and $m - 1$ respectively. Below we consider various algorithms A of a similar structure. We denote the size reduction factor, the number of multiplications, input adders, output adders, and the total number of bit operations to multiply two n-bit polynomials in

A by b_A, mul_A, ia_A, oa_A, and $\mathsf{M}_A(n)$, respectively. Then

$$\mathsf{M}_A(n) = \mathsf{mul}_A\ \mathsf{M}(m) + \mathsf{ia}_A\ m + \mathsf{oa}_A\ (2m-1) + 2(\mathsf{b}_A - 1)(m-1), \tag{3.2}$$

where $m = \lceil n/\mathsf{b}_A \rceil$ and $\mathsf{M}(m)$ is the cost of the multiplication call for m-bit polynomials. For $A = K_2$, this becomes:

$$\mathsf{M}_{K_2}(n) = 3\ \mathsf{M}(m) + 8m - 4, \quad m = \lceil n/2 \rceil.$$

Our interest is not the usual recursive deployment of this kind of algorithms, but rather the efficient interaction of various methods. We include in our study the classical multiplication C_b on b-bit polynomials and algorithms for 3, 5, 6, and 7-segment polynomials which we call K_3 (3-segment Karatsuba, see Blahut (1985), Section 3.4, page 85), M_5, M_6, and M_7 (see Montgomery (2005)). The parameters of these algorithms are given in Table 3.3.

Algorithm A	b_A	mul_A	ia_A	oa_A
K_2	2	3	2	2
K_3	3	6	6	6
M_5	5	13	22	30
M_6	6	17	61	40
M_7	7	22	21	55
$C_b, b \geq 2$	b	b^2	0	$(b-1)^2$

Table 3.3: The parameters of some multiplication methods

3.3 Hybrid Design

For fast multiplication software, a judicious mixture of table look-up and classical, Karatsuba and even faster (FFT) algorithms must be used (see von zur Gathen & Gerhard (2003), chapter 8, and Hankerson *et al.* (2003), chapter 2). Suitable techniques for hardware implementations are not thoroughly studied in the literature. In contrast to software implementations where the word-length of the processor, the datapath, and the set of commands are fixed, hardware designers have more flexibility. In software solutions speed and memory usage are the measures of comparison whereas hardware implementations are generally designed to minimize the area and time, simultaneously or with some weight-factors. In this section we determine the least-cost combination of any basic routines for bit sizes up to 8192. Here, cost corresponds to the total number of operations in software, and the area in hardware. Using pipelining and the structure of Grabbe *et al.* (2003a) this can also result in multipliers which have small area-time parameters.

We present a general methodology for this purpose. We start with a toolbox $\mathcal{T}$ of basic algorithms, namely $\mathcal{T} = \{\text{classical}, K_2, K_3, M_5, M_6, M_7\}$. Each $A \in \mathcal{T}$ is defined for b_A-bit polynomials. We denote by $\mathcal{T}^*$ the set of all iterated (or hybrid algorithms) compositions from $\mathcal{T}$; this includes $\mathcal{T}$, too.

Figure 3.4 shows the hierarchy of a hybrid algorithm for 12-bit polynomials using our toolbox $\mathcal{T}$. At the top level, K_2 is used, meaning that the 12-bit input polynomials are divided into two 6-bit polynomials each and K_2 is used to multiply the input polynomials as if each 6-bit polynomial were a single coefficient. K_2C_3 performs the three 6-bit multiplications. One of these 6-bit multipliers is circled in Figure 3.4 and unravels as follows:

$$
\begin{aligned}
&(a_5x^5 + \cdots + a_0) \cdot (b_5x^5 + \cdots + b_0) = \\
&((a_5x^2 + a_4x + a_3)x^3 + (a_2x^2 + a_1x + a_0)) \\
&\cdot ((b_5x^2 + b_4x + b_3)x^3 + (b_2x^2 + b_1x + b_0)) = \\
&(A_1x^3 + A_0) \cdot (B_1x^3 + B_0) = A_1B_1x^6 + \\
&((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)x^3 + A_0B_0
\end{aligned}
$$

Each of A_1B_1, $(A_1 + A_0)(B_1 + B_0)$, and A_0B_0 denotes a multiplication of 3-bit polynomials and will be done classically using the formula

$$
\begin{aligned}
&(a_2x^2 + a_1x + a_0)(b_2x^2 + b_1x + b_0) = a_2b_2x^4 + \\
&(a_2b_1 + a_1b_2)x^3 + (a_2b_0 + a_1b_1 + a_0b_2)x^2 + \\
&(a_1b_0 + a_0b_1)x + a_0b_0.
\end{aligned}
$$

Thick lines under each C_3 indicate the nine 1-bit multiplications to perform C_3. We designate this algorithm, for 12-bit polynomials, with $K_2K_2C_3 = K_2^2C_3$ where the left hand algorithm, in this case K_2, is the topmost algorithm.

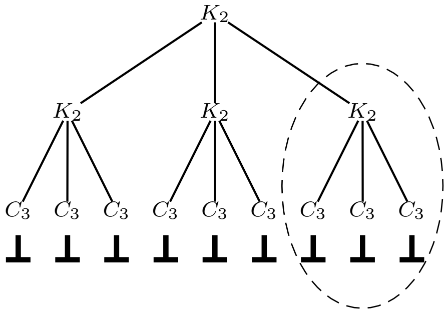


Figure 3.4: The multiplication hierarchy for $K_2K_2C_3$

As in (3.2), the cost of a hybrid algorithm $A_2A_1 \in \mathcal{T}^*$ with $A_1, A_2 \in \mathcal{T}^*$ satisfies

$$
\begin{aligned}
\mathsf{M}_{A_2A_1}(n) \leq & \mathsf{mul}_{A_2}\, \mathsf{M}_{A_1}(m) + \mathsf{ia}_{A_2}\, m + \\
& \mathsf{oa}_{A_2}\, (2m-1) + 2(\mathsf{b}_{A_2} - 1)(m-1),
\end{aligned}
\tag{3.3}
$$

where $\mathsf{M}_A(1) = 1$ for any $A \in \mathcal{T}^*$ and $m = \lceil n/(\mathsf{b}_{A_2}\mathsf{b}_{A_1}) \rceil = \lceil \lceil n/\mathsf{b}_{A_2} \rceil / \mathsf{b}_{A_1} \rceil$. Each $A \in \mathcal{T}^*$ has a well-defined input length b_A, given in Table 3.3 for basic tools and by multiplication for composite methods. We extend the notion by applying A also to fewer than b_A bits, by padding with leading zeros, so that $\mathsf{M}_A(m) = \mathsf{M}_A(\mathsf{b}_A)$ for $1 \leq m \leq \mathsf{b}_A$. For some purposes, one might consider the savings due to such a-priori-zero coefficients. Our goal, however, is a pipelined structure where such a consideration cannot be incorporated. The minimum hybrid cost over $\mathcal{T}$ is

$$
\mathsf{M}(n) = \min_{A \in \mathcal{T}^*, \mathsf{b}_A \geq n} \mathsf{M}_A(n).
$$

We first show that the infinitely many classical algorithms in $\mathcal{T}$ do not contribute to optimal methods beyond size 12.

Lemma 3. *For $A \in \mathcal{T}^*$ and integers $m \geq 1$ and $b, c \geq 2$ we have the following.*

(i) $\mathsf{M}_{C_b C_c}(bc) = \mathsf{M}_{C_{bc}}(bc)$.

(ii) $\mathsf{M}_{C_b A}(\mathsf{b}_A bm) \geq \mathsf{M}_{A C_b}(\mathsf{b}_A bm)$.

(iii) For any n, there is an optimal hybrid algorithm all of whose components are non-classical, except possibly the right most one.

(iv) If $n \geq 13$, then C_n is not optimal.

Proof. (*i*) This can be easily shown using (3.2) and Table 3.3.

(*ii*) We only show this for $A = K_2$. Using (3.2) and Table 3.3 we have

$$\mathsf{M}_{C_b K_2}(2bm) - \mathsf{M}_{K_2 C_b}(2bm) = 2(b-1)(3bm - b - 1) > 0.$$

(*iii*) Let $A = A_1 A_2 \cdots A_r$ be a hybrid algorithm with $A_1, \ldots, A_r \in \mathcal{T}$ and suppose that $A_s = C_b$ for some $s < r$ and $b \geq 2$ and $A_{s+1} \in \{K_2, \ldots, M_7\}$. Now (*ii*) shows that the cost of

$$A' = A_1 A_2 \cdots A_{s+1} A_s \cdots A_r$$

is smaller than that of A, and A is not optimal. Hence if some A_s is classical, then each A_t for $s < t \leq r$ is also classical. These can all be combined into one by (*i*).

(*iv*) We let $m = \lceil n/2 \rceil$. Then

$$M_{C_n}(n) - M_{K_2 C_m}(2m) =$$
$$2n^2 - 2n + 2 - 6m^2 - 2m + 2 \geq n^2/2 - 6n - 1/2 > 0$$

using $(n+1)/2 \geq m$ and $n \geq 13$. On the other hand, $n \leq 2m$ and the $2m$-bit algorithm $K_2 C_m$ can also be used for n-bit polynomials, and we have

$$\mathsf{M}_{K_2 C_m}(n) \leq \mathsf{M}_{K_2 C_m}(2m) < \mathsf{M}_{C_n}(n).$$

□

Algorithm 4 presents a dynamic programming algorithm which computes an optimal hybrid algorithm from $\mathcal{T}^*$ for n-bit multiplication, for $n = 1, 2, \ldots$.

Algorithm 4 Finding optimal algorithms in $\mathcal{T}^*$

Input: The toolbox $\mathcal{T} = \{\text{classical}, K_2, K_3, M_5, M_6, M_7\}$ and an integer N.

Output: Table T with N rows containing the optimal algorithms for $1 \leq n \leq N$ and their costs.

```
Enter the classical algorithm and cost 1 for $n = 1$ into $T$
for $n = 2, \ldots, N$ do
    $bestalgorithm \leftarrow$ unknown, $mincost \leftarrow$ +infinity
    for $A = K_2, \ldots, M_7$ do
        Compute $\mathsf{M}_A(n)$ according to (3.2)
        if $\mathsf{M}_A(n) < mincost$ then
            $bestalgorithm \leftarrow A$, $mincost \leftarrow \mathsf{M}_A(n)$
        end if
    end for
    if $n < 13$ then
        $\mathsf{M}_{C_n} \leftarrow 2n^2 - 2n + 1$
        if $\mathsf{M}_{C_n}(n) < mincost$ then
            $bestalgorithm \leftarrow C_n$, $mincost \leftarrow \mathsf{M}_{C_n}(n)$
        end if
    end if
    Enter $bestalgorithm$ and $mincost$ for $n$ into $T$
end for
```

Theorem 4. *Algorithm 4 works correctly as specified. The operations (arithmetic, table look-up) have integers with $O(\log N)$ bits as input, and their total number is $O(N)$.*

Proof. We only show correctness, by induction on n. The case $n = 1$ is clear. So let $n > 1$, and $A \in \mathcal{T}^*$ be an optimal algorithm for n-bit polynomials as in Lemma 3-(*iii*). We write $A = BC$ with $B \in \mathcal{T}$ and $C \in \mathcal{T}^*$. If B is non-classical, then it is tested for in steps 4-9, and by induction, an optimal algorithm D is chosen for the calls at size $m = \lceil n/\mathsf{b}_B \rceil$. Thus $\mathsf{M}_D(m) \leq \mathsf{M}_C(m)$ and in fact, equality holds. Therefore $\mathsf{M}_A(n) = \mathsf{M}_{BD}(n)$, and indeed an optimal algorithm BD is entered into T. If B is classical, then indeed $A = B$ and $n < 13$ by Lemma 3-(*iv*), and $A = C_n$ is tested in steps 10-14. □

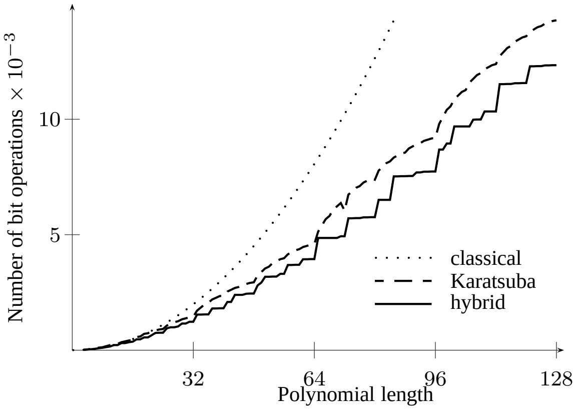


Figure 3.5: The number of bit operations of the classical, recursive Karatsuba, and the hybrid methods to multiply polynomials of degree smaller than 128

The optimal recursive method for each polynomial length up to 8192 is shown in Table 3.6. The column "length" of this table represents the length (or the range of lengths) of polynomials for which the method specified in the column "method" must be used. As an example, for 194-bit polynomials the method M_7 is used at the top level. This requires 22 multiplications of polynomials with $\lceil 194/7 \rceil = 28$ bits, which are done by means of K_2 on top of 14-bit polynomials. These 14-bit multiplications are executed again using K_2 and finally polynomials of length 7 are multiplied classically. Thus the optimal algorithm is $A = M_7 K_2^2 C_7$, of total cost $\mathsf{M}_A(194) = 22 \cdot \mathsf{M}_{K_2^2 C_7}(28) + 3937 = 26575$ bit operations.

Figure 3.5 shows the recursive cost of the Karatsuba method, as used by Weimerskirch & Paar (2003), of our hybrid method, and the classical method.

length	method	length	method	length	method
$1-5$	$C_1 - C_5$	$301-320$	K_2	$1603-1610$	M_5
6	K_2	$321-343$	M_7	$1611-1728$	M_6
7	C_7	$344-360$	M_5	$1729-1792$	M_7
8	K_2	$361-384$	K_2	$1793-1800$	M_5
9	K_3	$385-392$	M_7	$1801-1920$	M_6
10	K_2	$393-400$	M_5	$1921-1960$	M_7
11	C_{11}	$401-420$	M_7	$1961-2048$	K_2
$12-14$	K_2	$421-432$	K_2	$2049-2058$	M_7
15	K_3	$433-448$	M_7	$2059-2100$	M_5
$16-20$	K_2	$449-450$	M_5	$2101-2240$	M_7
21	M_7	$451-454$	K_2	$2241-2304$	M_6
$22-24$	K_2	455	M_5	$2305-2352$	M_7
25	M_5	456	K_2	$2353-2400$	M_6
$26-27$	K_3	$457-460$	M_5	$2401-2560$	K_2
$28-40$	K_2	$461-512$	K_2	$2561-2744$	M_7
$41-42$	M_7	$513-525$	M_5	$2745-2800$	M_5
$43-45$	K_3	$526-560$	M_7	$2801-2880$	M_6
$46-48$	K_2	$561-576$	K_2	$2881-3072$	K_2
49	M_7	$577-588$	M_7	$3073-3136$	M_7
50	M_5	$589-600$	M_5	$3137-3200$	M_5
$51-64$	K_2	$601-640$	K_2	$3201-3456$	M_6
$65-70$	M_7	$641-686$	M_7	$3457-3584$	M_7
$71-80$	K_2	$687-720$	M_5	$3585-3840$	M_6
$81-84$	M_7	$721-768$	K_2	$3841-3920$	M_7
$85-96$	K_2	$769-784$	M_7	$3921-4096$	K_2
$97-98$	M_7	$785-800$	M_5	$4097-4116$	M_7
$99-100$	M_5	$801-840$	M_7	$4117-4200$	M_5
$101-105$	M_7	$841-864$	M_6	$4201-4320$	M_6
$106-108$	K_2	$865-896$	M_7	$4321-4480$	M_7
$109-112$	M_7	$897-900$	M_5	$4481-4608$	M_6
$113-128$	K_2	$901-912$	M_6	$4609-4704$	M_7
$129-140$	M_7	$913-920$	M_5	$4705-4800$	M_6
$141-144$	K_2	$921-936$	M_6	$4801-5120$	K_2
$145-147$	M_7	$937-940$	M_5	$5121-5184$	M_6
$148-150$	M_5	$941-960$	M_6	$5185-5488$	M_7
$151-160$	K_2	$961-980$	M_7	$5489-5600$	M_5
$161-168$	M_7	$981-1024$	K_2	$5601-5880$	M_6
$169-175$	M_5	$1025-1029$	M_7	$5881-5888$	K_2
$176-192$	K_2	$1030-1050$	M_5	$5889-5952$	M_6
$193-196$	M_7	$1051-1120$	M_7	$5953-6016$	K_2
$197-200$	M_5	$1121-1152$	M_6	$6017-6144$	M_6
$201-210$	M_7	$1153-1176$	M_7	$6145-6272$	M_7
$211-216$	K_2	$1177-1200$	M_5	$6273-6400$	M_5
$217-224$	M_7	$1201-1280$	K_2	$6401-6912$	M_6
225	M_5	$1281-1372$	M_7	$6913-7168$	M_7
$226-256$	K_2	$1373-1440$	M_5	$7169-7680$	M_6
$257-280$	M_7	$1441-1536$	K_2	$7681-7840$	M_7
$281-288$	K_2	$1537-1568$	M_7	$7841-8064$	M_6
$289-294$	M_7	$1569-1600$	M_5	$8065-8192$	K_2
$295-300$	M_5	$1601-1602$	M_6		

Table 3.6: Optimal multiplications for polynomial lengths up to 8192

Lemma 3 implies that the classical methods need only be considered for $n \leq 12$. We can prune $\mathcal{T}$ further and now illustrate this for K_3. One first checks that $M_{AK_3B}(3\mathsf{b}_A\mathsf{b}_B) < M_{K_3AB}(3\mathsf{b}_A\mathsf{b}_B)$ for $A \in \{K_2, M_5, M_6, M_7\}$, $B \in \mathcal{T}^*$, and $\mathsf{b}_B \geq 2$. Therefore for K_3 to be the top-level tool in an optimal algorithm over $\mathcal{T}$ the next algorithm to it must be either K_3 or C_b for some b. Since the classical method is not optimal for $n \geq 13$ and Table 3.6 does not list K_3 in the interval 46 to $3 \cdot 45 = 135$, K_3 is not the top-level tool for $n \geq 135$.

Table 3.7 gives the asymptotic behavior of the costs of the algorithms in the toolbox $\mathcal{T}$ when used recursively. It is expected that for very large polynomials only the asymptotically fastest method, namely M_6, should be used. But due to the tiny differences in the cost exponents this seems to happen only for very large polynomial lengths, beyond the sizes which are shown in Table 3.6.

algorithm	k
$C_b, b \geq 2$	$\log_b b^2 = 2$
K_3	$\log_3 6 \approx 1.6309$
M_5	$\log_5 13 \approx 1.5937$
M_7	$\log_7 22 \approx 1.5885$
K_2	$\log_2 3 \approx 1.5850$
M_6	$\log_6 17 \approx 1.5812$

Table 3.7: Asymptotical cost $O(n^k)$ of algorithms in the toolbox $\mathcal{T}$

3.4 Hardware Structure

The delay of a fully parallel combinational Karatsuba multiplier is $4\lceil \log_2 n \rceil$, which is almost 4 times that of a classical multiplier, namely $\lceil \log_2 n \rceil + 1$. It is the main disadvantage of the Karatsuba method for hardware implementations. As a solution, we have suggested in Grabbe *et al.* (2003a) a pipelined Karatsuba multiplier for 240-bit polynomials, shown in Figure 3.8.

The innermost part of the design is a combinational pipelined 40-bit classical multiplier equipped with 40-bit and 79-bit adders. The multiplier, these adders, and the overlap

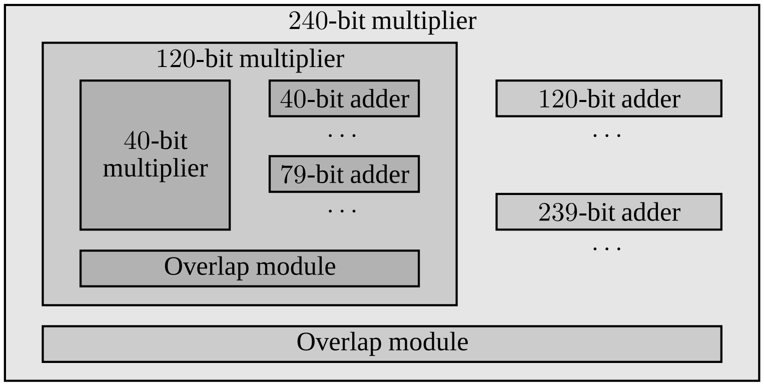


Figure 3.8: The 240-bit multiplier in Grabbe *et al.* (2003a)

module, together with a control circuit, constitute a 120-bit multiplier. The algorithm is based on a modification of a Karatsuba formula for 3-segment polynomials which is similar to but slightly different from what we have used in Section 3.3. Another suitable control circuit performs the 2-segment Karatsuba method for 240 bits by means of a 120-bit recursion, 239-bit adders, and an overlap circuit.

This multiplier can be seen as implementing the factorization $240 = 2 \cdot 3 \cdot 40$. Table 3.6 implies that it is preferable to use 2 or 5-segment for larger polynomials rather than the 3-segment method. On the other hand the complicated structure of the 5-segment method makes it difficult to use it for pipelining in the upper levels. A new design is presented here which is based on the factorization $240 = 2 \cdot 2 \cdot 2 \cdot 30$. The resulting structure is shown in Figure 3.9.

The 30-bit multiplier follows the recipe of Table 3.6. It is a combinational circuit without feedback and the design goal is to minimize its area. In general, k pipeline stages can perform n parallel multiplications in $n + k - 1$ instead of nk clock cycles without pipelining.

The new design, the structure of Grabbe *et al.* (2003a), and a purely classical method are designed on an XC2V6000-4FF1517-4 FPGA. The last design has a classical 30-bit multiplier and applies the three classical recursion steps to it. The results after place and route are shown in Table 3.10. The second column shows the number of clock cycles for

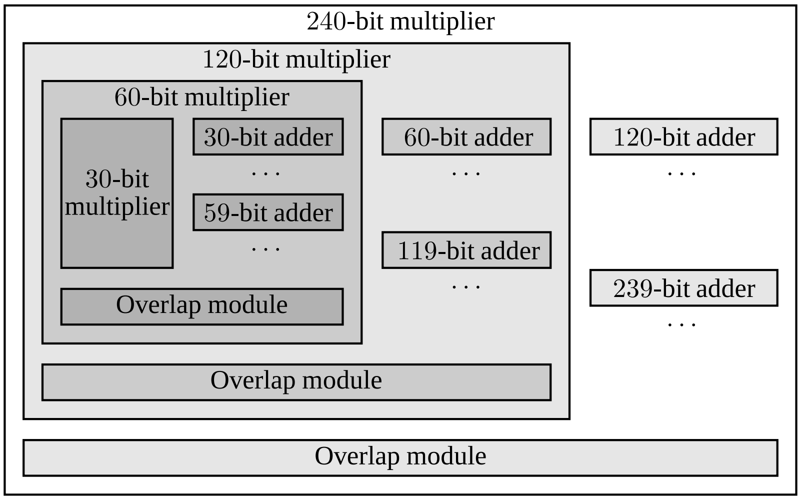


Figure 3.9: The new 240-bit multiplier

a multiplication. The third column represents the area in terms of the number of slices. This measure contains both logic elements, or LUTs, and flip-flops used for pipelining. The fourth column is the multiplication time as returned by the hardware synthesis tool. Finally the last column shows the product of area and time in order to compare the AT measures of our designs.

The synchronization is set so that the 30-bit multipliers require 1 and 4 clock cycles for classical and hybrid Karatsuba implementations, respectively. The new structure is smaller than the implementation in Grabbe *et al.* (2003a) but is larger than the classical one. This drawback is due to the complicated structure of the Karatsuba method but is compensated by the speed as seen in the time and AT measures. In the next section this is further improved by decreasing the number of recursions.

Multiplier type	**Number of clock cycles**	**Number of slices**	**Multiplication time**	**AT** Slices × μs
classical	106	1328	$1.029\mu s$	1367
Grabbe *et al.* (2003a) (Fig. 3.8)	54	1660	$0.655\mu s$	1087
Hybrid Karatsuba (Fig. 3.9)	55	1513	$0.670\mu s$	1014

Table 3.10: Time and area of different multipliers for 240-bit polynomials

3.5 Hybrid Polynomial Multiplier with Few Recursions

The timing diagram of the recursive multiplier for the time interval $50\,ns < t < 280\,ns$ is shown in Figure 3.11. In this figure the lowest level which is the combinational module communicates with the highest level through intermediate stages in several clock cycles. At time $t = 180\,ns$ after two 120-bit blocks being multiplied the 60-bit multiplier activates its *READY60* signal to inform the 120-bit multiplier to load it with new data. The 120-bit multiplier has to transfer this request to the 240-bit multiplier. Each of the multipliers require one clock cycle to start the lower multiplier. As a result it takes 5 clock cycles for the whole communication. This example reveals that in the recursive Karatsuba multiplier of Grabbe *et al.* (2003a), the core of the system, namely the combinational multiplier, is idle for several clock cycles during the multiplication. To improve resource usage, we reduce the communication overhead by decreasing the levels of recursion. In this new 240-bit multiplier, an 8-segment Karatsuba is applied at once to 30-bit polynomials. The formulas describing three recursive levels of Karatsuba are computed symbolically and implemented directly.

The new circuit is shown in Figure 3.12. The multiplexers *mux1* to *mux6* are adders at the same time. Their inputs are 30-bit sections of the two original 240-bit polynomials which are added according to the Karatsuba rules. Now their 27 output pairs are pipelined as inputs into the 30-bit multiplier. The 27 corresponding 59-bit polynomials are subsequently combined according to the overlap rules to yield the final result. Time and space consumptions are shown in Table 3.13 and compared with the results of Grabbe *et al.* (2003a). The columns are as in Table 3.10. It can be seen that this design improves on the previous ones in all respects.

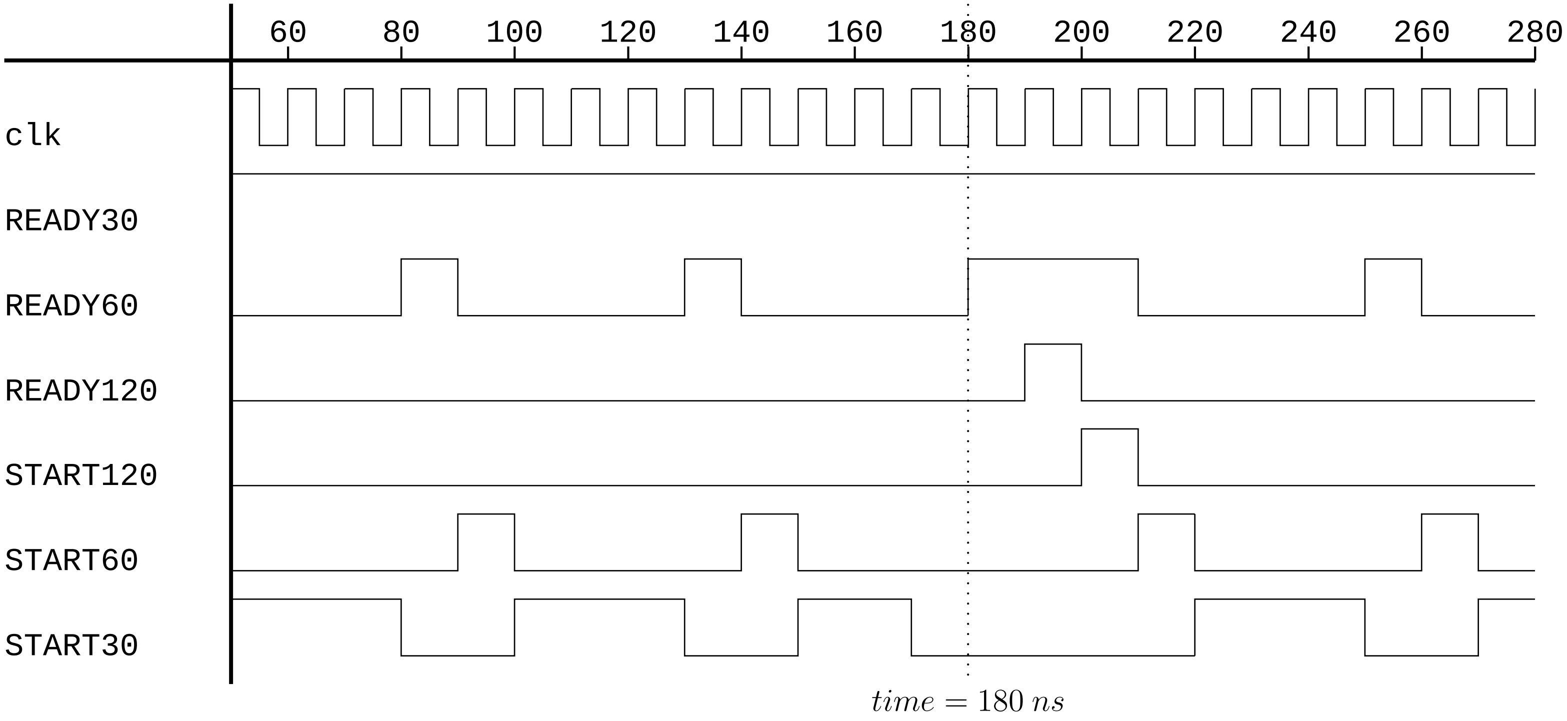


Figure 3.11: Timing diagram of the recursive Karatsuba multiplier for 240-bit polynomials in Figure 3.9

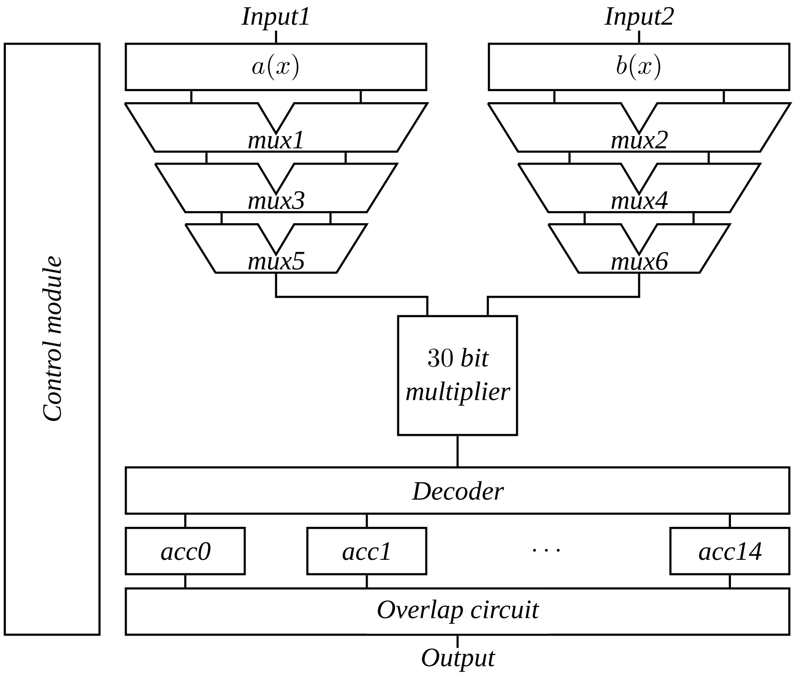


Figure 3.12: The structure of the Karatsuba multiplier with fewer number of recursions.

Multiplier type	**Number of clock cycles**	**Number of slices**	**Multiplication time**	**AT** Slices × μs
classical	65	1582	$0.523\mu s$	827
Grabbe *et al.* (2003a) (Fig. 3.8)	54	1660	$0.655\mu s$	1087
Fewer recursions (Fig. 3.12)	30	1480	$0.378\mu s$	559

Table 3.13: Time and area of different 240-bit multipliers compared with the structure with reduced number of recursion levels

3.6 Code Generator

As it has been shown in Sections 3.4 and 3.5, using a pipelined modular structure to implement the Karatsuba multiplication algorithm has several advantages. However, implementing the parallel multipliers for smaller polynomials is the main difficulty of this structure. Handwriting a VHDL code for these blocks is time consuming, particularly when accurate insertion of pipeline stages is desired.

In this section an object oriented library is presented which can be used for automatically generating the VHDL code of a combinational pipelined multiplier of any degree. This has been achieved by combining the Karatsuba and the classical methods. The generated code is in register transfer level (RTL).

3.6.1 Code Generator Functionalities

The code generator gives the user the ability of suppling a sequence of pairs of polynomial lengths and type selection parameters. The type selection parameter specifies the algorithm which is used for each polynomial degree, e.g., the Karatsuba or the classical algorithm. Then, the program generates the appropriate multiplication method for each of these degrees and combines them recursively to create the algorithm and consequently the multiplier.

An important functionality of the code generator is the ability of computing algorithms for multiplying polynomials, when an algorithm to multiply longer polynomials is found. The program automatically inserts zero coefficients to the beginnings of the smaller polynomials, makes the multiplication graphs, and removes the unnecessary gates.

The library is also able to report the time and space complexities of the design and create appropriate pipeline stages by getting the depth of each pipeline stage as the number of two input gates.

The main part of this code generator consists of the following classes. Each class is represented with its functionalities:

Multiplication

The class `multiplication` manages the multiplication methods. It creates the appropriate classical and Karatsuba methods, their shortenings and combines them recursively. This class is able to simplify the resulting expressions and to put pipeline registers in the appropriate positions. Finally, it generates a VHDL code which describes the multiplier.

An important functionality of the class `multiplication` is the generation of a computation graph for the polynomial multiplication. It computes the depth of the graph, puts pipeline registers according to a specified depth, checks for possible hazards, and increases the number of these registers when required to remove any hazards (see the section on pipelining below). Therefore, a pipelined multiplier can be generated in which the depth of the stages can be specified by the user as a parameter. A sample program which generates a multiplication method for polynomials of degree smaller than 6 is shown in Figure 3.14

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include "multiplication.hh"

int main(int argv, char* argc[]){

  multiplication m2;
  m2.init(); //Linear polynomials


  multiplication m6;
  m6.init();
  m6.expand(m2); //Cubic polynomials
  m6.shorten(3); //Quadratic polynomials
  m6.expand(m2); //Polynomials of degree 5

  m6.pipedepth = 4;
  m6.makecomputationsequence();

  m6.writeVHDL("mult.vhdl");
}
```

Figure 3.14: A sample code which uses the code generator libraries to produce a Karatsuba multiplier for polynomials of degree smaller than 6 when the pipeline depth is set to 4.

Addition Simplifier

The order of performing the additions in an expression has a great impact on the resource consumption. When the Karatsuba algorithm is used to multiply polynomials with large degrees, there are some additions which are redundant and can be performed only once. class `addition_simplifier` takes a set of additions and generates a specific sequence to compute it which contains only additions of two operands. This sequence can be optimized to achieve smaller area or shorter propagation delay. Achieving smaller area is done in a heuristic manner.

Reducing the Number of Gates

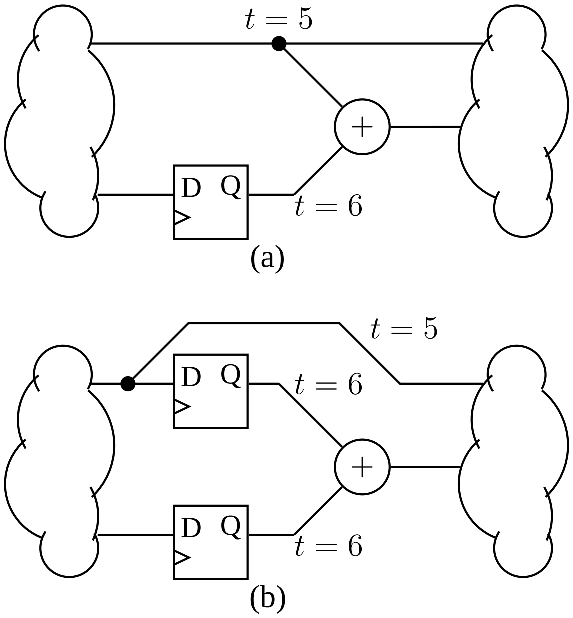
It has already been mentioned that the delay of small block multipliers does not have a large impact on the whole multiplication time. This happens when the number of independent multiplications is higher than the number of pipeline levels (which is often the case). Hence it is better to reduce the number of two input gates with the cost of increasing the propagation delay.

In order to identify and simplify redundant additions we count the number of simultaneous occurrences of each two variables. The two variables with the most number of occurrences are gathered together and represented with a new variable, which replaces all of their simultaneous occurrences. This is repeated until no two variables occur simultaneously in more than one expression.

Pipelining

The parallel combinational multipliers have complicated structures in which manually inserting the pipeline registers is a complicated task. Pipelining is an optimization technique which is used in the code generator. However, the pipeline depth must be supplied by the user as an input to the code generator. In an object of type *multiplication*, the sequence of operations is saved as a set of binary trees in which the position of pipeline registers are stored.

The most important issue in such a pipeline strategy is the *hazard* problem. It is possible that the inputs of a gate arrive in different time slices which will lead to failure in the computation of the result. To solve this problem, the pipeline generation module checks all the gates to see if the inputs arrive at the same time. If not, the path to the faster input will be delayed by an extra register. The Inputs to other modules will be taken from the former register as shown in Figure 3.15, in which registers are shown with boxes. This method solves the hazard problem without increasing the overall latency.



(a)

(b)

Figure 3.15: Pipelining the multiplier circuit (a) before and (b) after solving the *hazard* problem

3.7 Conclusion

Since finite field arithmetics play an important role in cryptography and elliptic curves, this chapter is devoted to the application of asymptotically fast methods, in particular the Karatsuba method in hardware. The materials of this chapter are presented in the following sequence:

- In Section 3.2 essentials of the Karatsuba method and its costs are analyzed.
- Section 3.3 proposes a new methodology which can be used to combine different multiplication methods of a toolbox to achieve a new algorithm. The new algorithm can be found in a time that is linear in the polynomial degree and its number of bit operations for each polynomial degree is at most equal to the number of bit operations for the best algorithm of the toolbox for that degree.
- Section 3.4 is devoted to the review of the multiplier of Grabbe *et al.* (2003a). The main contribution of this section is the application of the hybrid method to that multiplier and reducing the area in this way.
- In Section 3.5 fewer recursions are used to decrease the communication time between different modules in the modular Karatsuba multiplier. In this way multipliers are achieved which are better than the classical and the method of Grabbe *et al.* (2003a) with respect to both time and area.
- Finally Section 3.6 introduces a C++ based code generator by describing its functionalities and structure. This library is developed to generate the VHDL description of combinational pipelined multipliers and is used for the smaller multipliers of this chapter.

Chapter 4

Small Area Normal Basis Multipliers: Gauß meets Pascal

4.1 Introduction

Normal basis representation of finite fields enables easy computation of the qth power of elements. Assuming q to be a prime power, a basis of the form $(\alpha, \alpha^q, \cdots, \alpha^{q^{n-1}})$ for $\mathbb{F}_{q^n}$ is called a normal basis generated by the normal element $\alpha \in \mathbb{F}_{q^n}$. In this basis the qth power of an element can be computed by means of a single cyclic shift. This property makes such bases very attractive for parallel exponentiation in finite fields (see Nöcker (2001)).

Since multiplication in these bases is more expensive than in polynomial basis it is especially desirable to reduce their multiplication costs. In this chapter, a new method for multiplication in normal bases of type 2 is suggested. It uses an area efficient circuit to convert the normal basis representation to polynomials and vice versa. Any method can be used to multiply the resulting polynomials. Although this structure has small area, its propagation delay is longer than other methods and is only suitable for applications where the area is limited.

One popular normal basis multiplier is the Massey-Omura multiplier presented for the

2	3	5	11	23	29	41	53	83	89
113	131	173	179	191	233	239	251	281	293
359	419	431	443	491	509	593	641	653	659
683	719	743	761	809	911	953	1013	1019	1031
1049	1103	1223	1229	1289	1409	1439	1451	1481	1499
1511	1559	1583	1601	1733	1811	1889	1901	1931	1973
2003	2039	2063	2069	2129	2141	2273	2339	2351	2393
2399	2459	2543	2549	2693	2699	2741	2753	2819	2903
2939	2963	2969	3023	3299	3329	3359	3389	3413	3449
3491	3539	3593	3623	3761	3779	3803	3821	3851	3863
3911	4019	4073	4211	4271	4349	4373	4391	4409	4481
4733	4793	4871	4919	4943					

Table 4.1: The prime numbers $n < 5000$ for which $\mathbb{F}_{2^n}$ contains an optimal normal basis of type 2.

first time by Omura & Massey (1986). The space and time complexities of this multiplier increase with the number of nonzero coefficients in the matrix representation of the endomorphism $x \to \alpha x$ over $\mathbb{F}_{q^n}$, where α generates the normal basis. Mullin *et al.* (1989) show that this number is at least $2n - 1$ which can be achieved for optimal normal bases. Gao & Lenstra (1992) specify exactly the finite fields for which optimal normal bases exist. Relating these bases with the Gauss periods they grouped them into optimal normal bases of type 1 and 2 according to the Gauss periods used.

For security reasons only prime extension degrees are used in cryptography, whereas the extension degrees of the finite fields containing an optimal normal basis of type 1 are always composite numbers. Cryptography standards often suggest the finite fields for which the type of normal bases are small (see for example FIPS PUB 186-2 (2000)) to enable designers to deploy normal bases. Table 4.1 shows the prime numbers n, when $n < 5000$, for which $\mathbb{F}_{2^n}$ contains an optimal normal basis of type 2. Applications in cryptography have stimulated research about efficient multiplication using optimal normal bases of type 2. The best space complexity results for the type 2 multipliers are n^2 and $3n(n-1)/2$ gates of types AND and XOR, respectively reported in Sunar & Koç (2001) and Reyhani-Masoleh & Hasan (2002). Their suggested circuits are obtained by suitably modifying the Massey-Omura multiplier. A classical polynomial basis multiplier, however, requires n^2 and $(n-1)^2$ gates of types AND and XOR respectively for the polynomial multiplication, followed by a modular reduction. The latter is done using a small circuit of size of $(r-1)n$, where r is the number of nonzero coefficients in the polynomial which is used to create the polynomial basis. It is conjectured by von zur Gathen & Nöcker (2005) that there are usually irreducible trinomials of degree n and for the cases that there is no irreducible trinomial an irreducible pentanomial can be found. The above costs and the fact that there are asymptotically fast methods for polynomial arithmetic suggest the use of polynomial multipliers for normal bases to make good use of both representations. The proposed multiplier in this chapter works in normal bases but its space complexity is similar to polynomial multipliers. Using classical polynomial multiplication methods, it requires $2n^2 + 16n\log_2(n)$ gates in $\mathbb{F}_{2^n}$. Moreover, using more efficient polynomial multiplication algorithms, such as the Karatsuba algorithm, we can decrease the space asymptotically even further down to $O(n^{\log_2 3})$.

The connection between polynomial and normal bases, together with its application in achieving high performance multiplication in normal bases, has been investigated in Gao *et al.* (1995) and Gao *et al.* (2000). The present work can be viewed as a conceptual continuation of the approach in those papers. Gao *et al.* (2000) describe how the multiplication using the normal bases generated by the Gauss periods can be reduced to multiplications of polynomials. For the case of the Gauss periods of type $(n, 2)$, their proposed method requires multiplication of two $2n$-bit polynomials which will be done using asymptotically fast methods, as suggested in their works.

The multiplier of this chapter is based on a similar approach. For optimal normal bases of type 2 we present a very efficient method which changes the representations between the normal basis and suitable polynomials. These polynomials are multiplied using any method of choice, such as the classical or the Karatsuba multiplier. Using the inverse transformation circuit and an additional small circuit the result is converted back into the normal basis representation. The heart of this method is a factorization of the transformation matrix between the two representations into a small product of sparse matrices. The circuit requires roughly $O(n \log n)$ gates and resembles the circuit used for computing the Fast Fourier Transformation (FFT). The analogy to the FFT circuit goes even further: as with the FFT, the inverse of the transformation has a very similar circuit. It should be noted that a general basis conversion, and not for a specific set of bases, requires $O(n^2)$ operation as also reported by Kaliski & Liskov (1999).

This chapter will begin with a review of the Gauss periods and the normal bases of type 2. Then the structure of the multiplier is introduced and the costs of each part of the multiplier are computed. The last section focuses the results on fields of characteristic 2 and compares the results with the literature.

4.2 Gauss Periods

Let $n, k \geq 1$ be integers such that $r = nk + 1$ is a prime, and let q be a prime power which is relatively prime to r. Then the group $\mathbb{Z}_r^\times$ of units modulo r is cyclic, has nk elements, and since $q^{nk} \equiv 1 \mod r$, r divides $q^{nk} - 1 = \#\mathbb{F}_{q^{nk}}^\times$. Hence there exists a primitive rth root of unity $\beta \in \mathbb{F}_{q^{nk}}$. Let $\mathcal{G} < \mathbb{Z}_r^\times$ be the unique subgroup of the cyclic group $\mathbb{Z}_r^\times$ with $\#\mathcal{G} = k$, and:

$$\alpha = \sum_{a \in \mathcal{G}} \beta^a$$

Then α is called a prime *Gauss period* of type (n, k) over $\mathbb{F}_q$. Wassermann (1993) and Gao *et al.* (2000) prove the following theorem:

Theorem 5. *Let $r = nk + 1$ be a prime not dividing q, e the index of q in $\mathbb{Z}_r^\times$, $\mathcal{G}$ the unique subgroup of order k of $\mathbb{Z}_r^\times$, and β a primitive rth root of unity in $\mathbb{F}_{q^r}$. Then the Gauss period*

$$\alpha = \sum_{a \in \mathcal{G}} \beta^a$$

is a normal element in $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ if and only if $\gcd(e, n) = 1$.

In this chapter we consider the Gauss periods of type $(n, 2)$. In this case $\mathcal{G} = \{1, -1\}$ and α is of the form $\beta + \beta^{-1}$, where β is a $2n+1$st root of unity in $\mathbb{F}_q^{2n}$. Hence the normal basis is

$$\mathcal{N} = (\beta + \beta^{-1}, \beta^q + \beta^{-q}, \cdots, \beta^{q^{n-1}} + \beta^{-q^{n-1}}) \tag{4.1}$$

and as is shown in Wassermann (1993) and Gao *et al.* (2000):

$$\{1, -1, q, -q, \cdots, q^{n-1}, -q^{n-1}\} = \{1, -1, 2, -2, \cdots, n, -n\}$$

if the computations are modulo $2n + 1$. Since $\beta^{2n+1} = 1$ each $\beta^{q^r} + \beta^{-q^r}$, $0 \leq r < n$, equals $\beta^i + \beta^{-i}$ for a suitable value of i, where $1 \leq i \leq n$. It follows that the normal basis representation

$$\sum_{k=0}^{n-1} a_k^{(\mathcal{N})}(\beta^{q^k} + \beta^{-q^k})$$

can be written as:

$$\sum_{l=1}^{n} a_l^{(\mathcal{N}')}(\beta^l + \beta^{-l}), \tag{4.2}$$

where $(a_l^{(\mathcal{N}')})_{1 \leq l \leq n}$ is a permutation of $(a_k^{(\mathcal{N})})_{0 \leq k < n}$. We call the sequence

$$\mathcal{N}' = (\beta + \beta^{-1}, \beta^2 + \beta^{-2}, \cdots, \beta^n + \beta^{-n}),$$

in this case, the permuted normal basis and the vector $(a_l^{(\mathcal{N}')})_{1 \leq l \leq n}$ the permuted normal representation of a.

4.3 Multiplier Structure

The structure of the multiplier is described in Figure 4.2. To multiply two elements $a, b \in \mathbb{F}_{q^n}$ given in the basis (4.1) we first convert their representations to the permuted form as

$$a = \sum_{i=1}^{n} a_i^{(\mathcal{N}')}(\beta^i + \beta^{-i}), \text{ and} \qquad b = \sum_{i=1}^{n} b_i^{(\mathcal{N}')}(\beta^i + \beta^{-i}),$$

with $a_i^{(\mathcal{N}')}, b_i^{(\mathcal{N}')} \in \mathbb{F}_q$. By inserting a zero at the beginning of the representation vectors and a linear mapping π_{n+1}, which we define in Section 4.4, from $\mathbb{F}_q^{n+1}$ to $\mathbb{F}_q[x]^{\leq n}$ the vectors of these representations are converted to polynomials $\varphi_a(x)$ and $\varphi_b(x)$ such that the evaluations of these two polynomials at $\beta+\beta^{-1}$ are a and b, respectively. The polynomials φ_a and φ_b are then multiplied using an appropriate method with respect to the polynomial degrees and implementation platform. Obviously the evaluation of the resulting polynomial $\varphi_c(x)$ at $\beta+\beta^{-1}$ is the product $c = a \cdot b$. The polynomial $\varphi_c(x)$ is of degree at most $2n$ and the evaluation is a linear combination of $(\beta+\beta^{-1})^i$ for $0 \leq i \leq 2n$. Using another linear mapping ν_{2n+1} from $\mathbb{F}_q[x]^{\leq 2n}$ to $\mathbb{F}_q^{2n+1}$, namely the inverse of π_{2n+1}, this linear combination is converted to a linear combination of the vectors 1 and $\beta^i+\beta^{-i}$ for $1 \leq i \leq 2n$. This is then converted to the permuted normal basis using another linear mapping τ_n.

The linear mapping ν_{2n+1} takes a polynomial in $\mathbb{F}_q[x]^{\leq 2n}$, evaluates it at $\beta+\beta^{-1}$, and represents the result as a linear combination of 1 and $\beta^i+\beta^{-i}$, for $1 \leq i \leq 2n$. Since the above vectors are linearly dependent there are several choices for ν_{2n+1}. One way to compute the resulting linear combination is that each $(\beta+\beta^{-1})^j$, for $1 \leq j \leq 2n$ be expanded as a linear combination of $\beta^i+\beta^{-i}$, for $1 \leq i \leq 2n$. The coefficients of these expansions have tight connections with the binomial coefficients or the entries of the Pascal triangle. Since $\beta \in \mathbb{F}_{q^{2n}}$, if we denote the characteristic of $\mathbb{F}_q$ by p, we have $p{\cdot}\beta = 0$ and the matrix representation of ν_{2n+1} has a similar structure as the Pascal triangle in which the entries are reduced modulo p. Such a triangle which has a fractal structure has attracted a lot of attention and has been given various names in the literature. One of the most famous ones is “Sierpinski triangle” or “Sierpinski gasket” (see Wikipedia (2006)) for $p = 2$. In Section 4.5 we find a special factorization for the matrix representation of ν_{2n+1} in an appropriate basis which allows the mapping to be computed in $O(n \log n)$ operations.

4.3.1 Example over $\mathbb{F}_{2^5}$

Here the overall operation of the multiplier for $\mathbb{F}_{2^5}$ is exemplified. Since 11 divides $2^{10}-1$ there is an 11th root of unity in $\mathbb{F}_{2^{10}}$ which is represented by β. Setting r, k, and q of

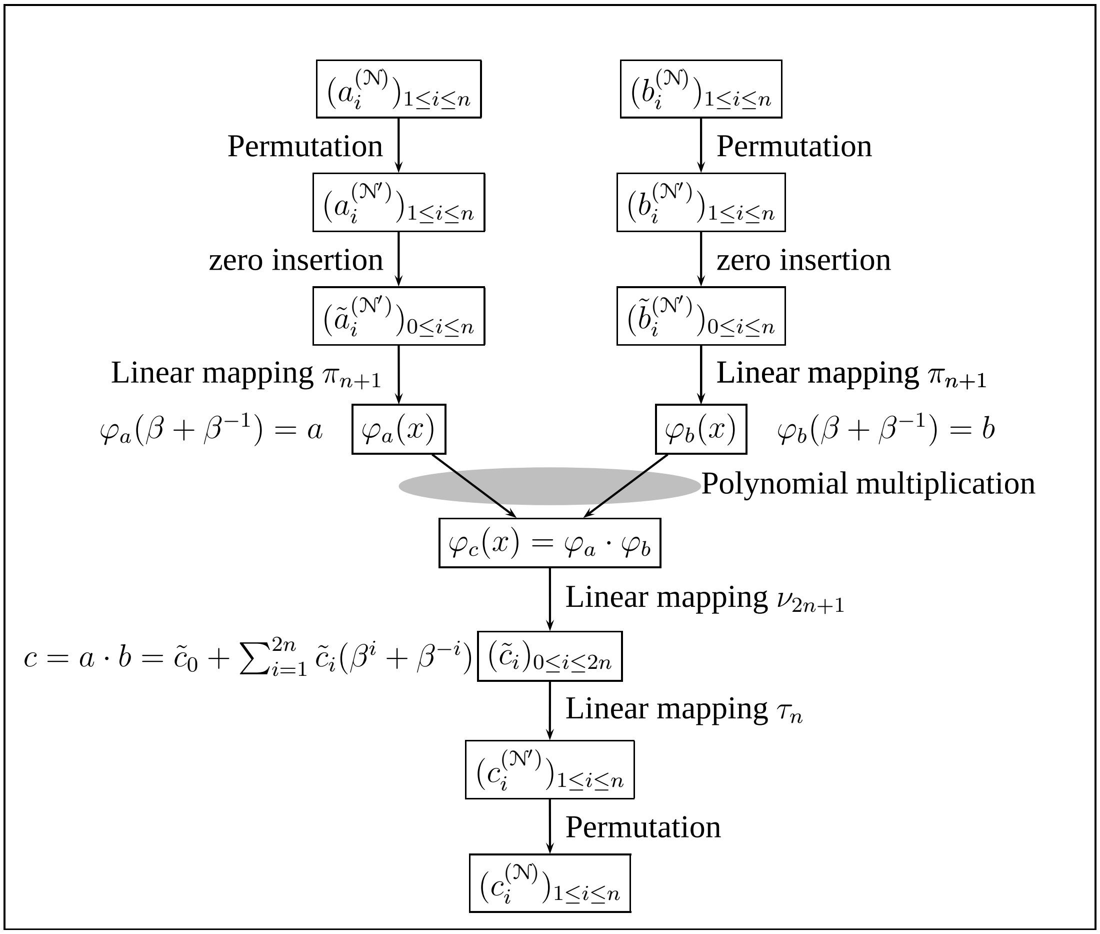

$(a_i^{(\mathcal{N})})_{1\le i\le n}$
$(b_i^{(\mathcal{N})})_{1\le i\le n}$
Permutation
Permutation
$(a_i^{(\mathcal{N}')})_{1\le i\le n}$
$(b_i^{(\mathcal{N}')})_{1\le i\le n}$
zero insertion
zero insertion
$(\tilde{a}_i^{(\mathcal{N}')})_{0\le i\le n}$
$(\tilde{b}_i^{(\mathcal{N}')})_{0\le i\le n}$
Linear mapping π_{n+1}
Linear mapping π_{n+1}
$\varphi_a(\beta+\beta^{-1})=a$
$\varphi_a(x)$
$\varphi_b(x)$
$\varphi_b(\beta+\beta^{-1})=b$
Polynomial multiplication
$\varphi_c(x)=\varphi_a\cdot\varphi_b$
Linear mapping ν_{2n+1}
$c=a\cdot b=\tilde{c}_0+\sum_{i=1}^{2n}\tilde{c}_i(\beta^i+\beta^{-i})$
$(\tilde{c}_i)_{0\le i\le 2n}$
Linear mapping τ_n
$(c_i^{(\mathcal{N}')})_{1\le i\le n}$
Permutation
$(c_i^{(\mathcal{N})})_{1\le i\le n}$

Figure 4.2: Overview of our multiplier structure to multiply two elements $a, b \in \mathbb{F}_{q^n}$, where their two representation vectors $(a_i^{(\mathcal{N})})_{1\le i\le n}$ and $(b_i^{(\mathcal{N})})_{1\le i\le n}$ with respect to the normal basis $\mathcal{N}$ are given. See the text for more information.

Theorem 5 to 11, 2, and 2, respectively, implies that $\alpha = \beta + \beta^{-1}$ constructs the normal basis $\mathcal{N}$ for $\mathbb{F}_{2^5}$ over $\mathbb{F}_2$:

$$\mathcal{N} = (\alpha, \alpha^2, \alpha^{2^2}, \alpha^{2^3}, \alpha^{2^4}).$$

Since β is an 11th root of unity the following equalities hold:

$$\begin{array}{lll} \beta+\beta^{-1}=\beta+\beta^{-1} & \beta^2+\beta^{-2}=\beta^2+\beta^{-2} & \beta^{2^2}+\beta^{-2^2}=\beta^4+\beta^{-4} \\ \beta^{2^3}+\beta^{-2^3}=\beta^3+\beta^{-3} & \beta^{2^4}+\beta^{-2^4}=\beta^5+\beta^{-5}. & \end{array} \tag{4.3}$$

and

$$\mathcal{N}' = (\beta+\beta^{-1}, \beta^2+\beta^{-2}, \beta^3+\beta^{-3}, \beta^4+\beta^{-4}, \beta^5+\beta^{-5})$$

is the *permuted normal* basis. We represent the vectors of the permuted normal representations of a and b by $\mathfrak{a}_{\mathcal{N}'} = (a_i^{(\mathcal{N}')})_{1 \le i \le 5}$ and $\mathfrak{b}_{\mathcal{N}'} = (b_i^{(\mathcal{N}')})_{1 \le i \le 5}$, respectively. These vectors satisfy the equations:

$$a = \mathcal{N}' \cdot \mathfrak{a}_{\mathcal{N}'}^T, \text{ and } \quad b = \mathcal{N}' \cdot \mathfrak{b}_{\mathcal{N}'}^T. \tag{4.4}$$

Our strategy is to find polynomials $\varphi_a(x)$ and $\varphi_b(x)$ over $\mathbb{F}_q$ whose evaluations at $\beta+\beta^{-1}$ give the elements a and b, respectively. Then these polynomials are multiplied and the evaluation of the result at $\beta+\beta^{-1}$ is converted back to the normal basis representation.

Each power $(\beta+\beta^{-1})^j$ can be represented as a linear combination of $\beta^i+\beta^{-i}$, for $0 \le i \le j$, in which the coefficients are in $\mathbb{F}_2$. Hence we have the following equality:

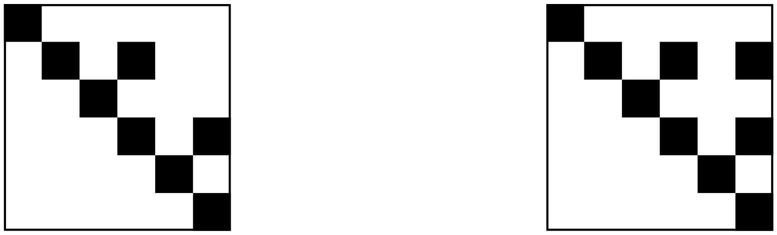
$$\mathcal{P} = \tilde{\mathcal{N}}' \cdot L_6, \tag{4.5}$$

in which:

$$\begin{aligned} \tilde{\mathcal{N}}' &= (1, \beta+\beta^{-1}, \beta^2+\beta^{-2}, \beta^3+\beta^{-3}, \beta^4+\beta^{-4}, \beta^5+\beta^{-5}), \\ \mathcal{P} &= (1, \beta+\beta^{-1}, (\beta+\beta^{-1})^2, (\beta+\beta^{-1})^3, (\beta+\beta^{-1})^4, (\beta+\beta^{-1})^5), \end{aligned}$$

and the matrix L_6, whose entries are calculated using the binomial coefficients modulo 2, is shown in part (a) of Figure 4.3. This matrix is upper triangular and hence invertible. We represent its inverse, which is shown in part (b) of Figure 4.3, by P_6, then:

$$\tilde{\mathcal{N}}' = \mathcal{P} \cdot P_6. \tag{4.6}$$



(a) (b)

Figure 4.3: (a) The matrix L_6 and (b) its inverse P_6

The entries of the vectors $\tilde{\mathcal{N}}'$ and $\mathcal{P}$ are elements of $\mathbb{F}_{2^5}$, which has dimension 5 over $\mathbb{F}_2$. These entries in each of the vectors are linearly dependent but still span $\mathbb{F}_{2^5}$ and each element of $\mathbb{F}_{2^5}$ can be written as a linear combination – which is not unique – of these elements. To represent a and b with respect to these new vectors we insert a zero at the beginning of the permuted normal representations to get the new vectors $\mathfrak{a}_{\tilde{\mathcal{N}}'} = (\tilde{a}_i^{(\tilde{\mathcal{N}}')})_{0\le i\le 5}$ and $\mathfrak{b}_{\tilde{\mathcal{N}}'} = (\tilde{b}_i^{(\tilde{\mathcal{N}}')})_{0\le i\le 5}$, respectively, i.e.,

$$\tilde{a}_i^{(\tilde{\mathcal{N}}')} = \begin{cases} 0 & \text{if } i = 0, \\ a_i^{(\mathcal{N}')} & \text{otherwise,} \end{cases} \qquad \tilde{b}_i^{(\tilde{\mathcal{N}}')} = \begin{cases} 0 & \text{if } i = 0, \\ b_i^{(\mathcal{N}')} & \text{otherwise.} \end{cases}$$

Similar to (4.4) we have:

$$a = \tilde{\mathcal{N}}' \cdot \mathfrak{a}_{\tilde{\mathcal{N}}'}^T, \text{ and } \quad b = \tilde{\mathcal{N}}' \cdot \mathfrak{b}_{\tilde{\mathcal{N}}'}^T. \tag{4.7}$$

Substituting $\tilde{\mathcal{N}}'$ from (4.6) we have:

$$a = \mathcal{P} \cdot P_6 \cdot \mathfrak{a}_{\tilde{\mathcal{N}}'}^T, \text{ and } \quad b = \mathcal{P} \cdot P_6 \cdot \mathfrak{b}_{\tilde{\mathcal{N}}'}^T. \tag{4.8}$$

Now consider the two vectors $\mathfrak{a}_{\mathcal{P}} = P_6 \cdot \mathfrak{a}_{\tilde{\mathcal{N}}'}^T = (a_i^{(\mathcal{P})})_{0\le i\le 5}$ and $\mathfrak{b}_{\mathcal{P}} = P_6 \cdot \mathfrak{b}_{\tilde{\mathcal{N}}'}^T = (b_i^{(\mathcal{P})})_{0\le i\le 5}$. They correspond to the two polynomials $\varphi_a(x) = \sum_{i=0}^{5} a_i^{(\mathcal{P})} x^i$ and $\varphi_b(x) = \sum_{i=0}^{5} b_i^{(\mathcal{P})} x^i$ whose evaluations at $\beta + \beta^{-1}$, according to (4.8), give a and b, respectively. Let the polynomial $\varphi_c(x) = \varphi_a(x) \cdot \varphi_b(x)$ be

$$\varphi_c(x) = \sum_{i=0}^{10} c_i^{(\mathcal{P})} x^i$$

and we have

$$c = a \cdot b = \sum_{i=0}^{10} c_i^{(\mathcal{P})} (\beta + \beta^{-1})^i.$$

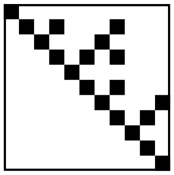


Figure 4.4: The matrix L_{11}

In the same way as above we can use the matrix L_{11} which corresponds $((\beta+\beta^{-1})^i)_{0\le i\le 10}$ with the vector containing 1 and $\beta^i+\beta^{-i}$ for $1 \le i \le 10$. The matrix L_{11} is shown in Figure 4.4. The product of L_{11} and $\mathfrak{c}_{\mathcal{P}} = (c_i^{(\mathcal{P})})_{0\le i\le 10}$ is a new vector $\mathfrak{c}_{\tilde{\mathcal{N}}'} = (\tilde{c}_i^{(\tilde{\mathcal{N}}')})_{0\le i\le 10}$:

$$\mathfrak{c}_{\tilde{\mathcal{N}}'} = L_{11}\cdot \mathfrak{c}_{\mathcal{P}}^T$$

and we have:

$$c = \tilde{c}_0^{(\tilde{\mathcal{N}}')} + \sum_{i=1}^{10} \tilde{c}_i^{(\tilde{\mathcal{N}}')}(\beta^i+\beta^{-i}).$$

As it will be seen later $\tilde{c}_0^{(\tilde{\mathcal{N}}')}$ for fields of characteristic 2 is always zero. For other fields we need to compute the representation of 1 and multiply it by $\tilde{c}_0^{(\tilde{\mathcal{N}}')}$. On the other hand since $\beta^{11}=1$ we have

$$\begin{array}{ll}\beta^6+\beta^{-6}=\beta^5+\beta^{-5}, & \beta^7+\beta^{-7}=\beta^4+\beta^{-4},\\ \beta^8+\beta^{-8}=\beta^3+\beta^{-3}, & \beta^9+\beta^{-9}=\beta^2+\beta^{-2}, \text{ and}\\ \beta^{10}+\beta^{-10}=\beta+\beta^{-1}. & \end{array}$$

Also the permuted normal and normal representations of the product are

$$(\tilde{c}_1^{(\tilde{\mathcal{N}}')}+\tilde{c}_{10}^{(\tilde{\mathcal{N}}')}, \tilde{c}_2^{(\tilde{\mathcal{N}}')}+\tilde{c}_9^{(\tilde{\mathcal{N}}')}, \tilde{c}_3^{(\tilde{\mathcal{N}}')}+\tilde{c}_8^{(\tilde{\mathcal{N}}')}, \tilde{c}_4^{(\tilde{\mathcal{N}}')}+\tilde{c}_7^{(\tilde{\mathcal{N}}')}, \tilde{c}_5^{(\tilde{\mathcal{N}}')}+\tilde{c}_6^{(\tilde{\mathcal{N}}')}) \text{ and}$$
$$(\tilde{c}_1^{(\tilde{\mathcal{N}}')}+\tilde{c}_{10}^{(\tilde{\mathcal{N}}')}, \tilde{c}_2^{(\tilde{\mathcal{N}}')}+\tilde{c}_9^{(\tilde{\mathcal{N}}')}, \tilde{c}_4^{(\tilde{\mathcal{N}}')}+\tilde{c}_7^{(\tilde{\mathcal{N}}')}, \tilde{c}_3^{(\tilde{\mathcal{N}}')}+\tilde{c}_8^{(\tilde{\mathcal{N}}')}, \tilde{c}_5^{(\tilde{\mathcal{N}}')}+\tilde{c}_6^{(\tilde{\mathcal{N}}')}),$$

respectively. In the next sections we compute the costs of each of the above tasks for general p and n.

4.4 Polynomials from Normal Bases

The most important parts of the multiplier are the converters between polynomial and permuted normal representations. Since the elements $(\beta+\beta^{-1})^i$, for $0 \leq i \leq n$, and also 1 and $\beta^i+\beta^{-i}$, for $1 \leq i \leq 2n$, are linearly dependent there are different possibilities for the selection of the mappings π_{n+1} and ν_{2n+1} from Section 4.3. These mappings are defined via matrices $P_{n+1} \in \mathbb{F}_p^{(n+1)\times(n+1)}$ and $L_{2n+1} \in \mathbb{F}_p^{(2n+1)\times(2n+1)}$, where p is the characteristic of $\mathbb{F}_{q^n}$. These matrices have special factorizations which let them to be multiplied by vectors of appropriate length using $O(n \log n)$ operations in $\mathbb{F}_q$.

The idea behind the construction of these matrices is similar to the example in Section 4.3. The permuted representations of a and b are preceeded by zero and P_{n+1} is multiplied by the resulting vectors. The structure of the inverse of P_{n+1} which we denote by L_{n+1} is easier to describe. Hence we define a candidate for L_{n+1} and show that this matrix can be used to convert from polynomial to the extended permuted normal representation, i.e., it satisfies

$$\begin{array}{l}(1, \beta+\beta^{-1}, \beta^2+\beta^{-2}, \cdots, \beta^n+\beta^{-n})L_{n+1} = \\ (1, \beta+\beta^{-1}, (\beta+\beta^{-1})^2, \cdots, (\beta+\beta^{-1})^n).\end{array}$$

We also show that L_{n+1} is invertible. Then we study its structure and show how it can be factored into sparse factors in Section 4.5. This factorization is also used to find a factorization for P_n.

Definition 6. *Let p be the characteristic of $\mathbb{F}_q$, for integers i, j let $l_{i,j} \in \mathbb{F}_p$ be such that $(x+x^{-1})^j = \sum_{i\in\mathbb{Z}} l_{i,j}x^i$ in $\mathbb{F}_p[x]$, for a variable x, and $L_{q,n} = (l_{i,j})_{0\leq i,j<n} \in \mathbb{F}_p^{n\times n}$. Obviously $l_{i,j} = 0$ for $|i| > |j|$. ($L_{q,n}$ depends on p but not on $\log_p q$.)*

Example 7. *Let $q = 9$, i.e., $p = 3$. For $0 \leq j < 9$, the powers $(x+x^{-1})^j$ are:*

j	$(x+x^{-1})^j$
0	1
1	$x+x^{-1}$
2	x^2+2+x^{-2}
3	x^3+x^{-3}
4	$x^4+x^2+x^{-2}+x^{-4}$
5	$x^5+2x^3+x+x^{-1}+2x^{-3}+x^{-5}$
6	x^6+2+x^{-6}
7	$x^7+x^5+2x+2x^{-1}+x^{-5}+x^{-7}$
8	$x^8+2x^6+x^4+2x^2+1+2x^{-2}+x^{-4}+2x^{-6}+x^{-8}$.

Hence the matrix $L_{9,9}$ is:

$$\begin{pmatrix} 1 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Theorem 8. *The matrix $L_{q,n}$ of Definition 6 satisfies*

$$\begin{array}{l}(1,\beta+\beta^{-1},\beta^2+\beta^{-2},\cdots,\beta^{n-1}+\beta^{-n+1})L_{q,n}= \\ (1,\beta+\beta^{-1},(\beta+\beta^{-1})^2,\cdots,(\beta+\beta^{-1})^{n-1}),\end{array} \tag{4.9}$$

is upper triangular with 1 on the diagonal, hence nonsingular, and its entries satisfy the relation:

$$(L_{q,n})_{i,j}=\begin{cases} 0 & \textit{if } i>j \textit{ or } j-i \textit{ is odd, and} \\ \binom{j}{(j-i)/2} & \textit{otherwise.}\end{cases}$$

Proof. Since $(x+x^{-1})^j=(x^{-1}+x)^j$ we have $l_{i,j}=l_{-i,j}$ and for any $0\le j<n$:

$$(\beta+\beta^{-1})^j=\sum_{i=-j}^{j} l_{i,j}\beta^i = l_{0,j}+\sum_{i=1}^{j} l_{i,j}(\beta^i+\beta^{-i}) = l_{0,j}+\sum_{i=1}^{n-1} l_{i,j}(\beta^i+\beta^{-i}),$$

since $l_{i,j} = 0$ for $i > j$. This shows that the jth entries on the left and right sides of (4.9) for $0 \leq j < n$ are equal. For the values of $(L_{q,n})_{i,j}$ we have:

$$(x + x^{-1})^j = \sum_{k=0}^{j} \binom{j}{k} x^{j-k}(x^{-1})^k = \sum_{k=0}^{j} \binom{j}{k} x^{j-2k}, \tag{4.10}$$

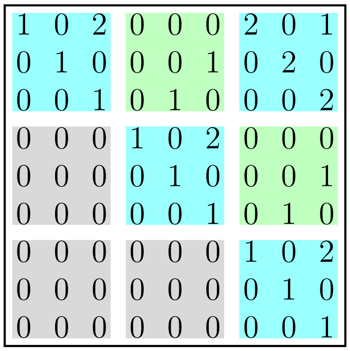
in which the binomial coefficients are reduced modulo p and the coefficient of x^i is the entry $(L_{q,n})_{i,j}$. All of the powers of x in (4.10) when $i > j$ have zero coefficients. For the remaining terms if $i - j$ is odd there is no integer k such that $i = j - 2k$, hence the entry $(L_{q,n})_{i,j}$ is zero. For even values $i = j - 2k$ implies that $k = (j - i)/2$ and $(L_{q,n})_{i,j}$ is $\binom{j}{(j-i)/2}$. Since $L_{q,n}$ is upper triangular its determinant is the product of all elements on the main diagonal. The entry $(L_{q,n})_{i,j}$ is $\binom{j}{0} = 1$ and the determinant is also equal to 1. □

Definition 9. *Let $L_{q,n}$ be as defined in Definition 6. We denote its inverse by $P_{q,n} = (p_{i,j})_{0 \leq i,j < n}$, where $p_{i,j} \in \mathbb{F}_p$ and p is the characteristic of $\mathbb{F}_q$.*

As we have seen the entries of the matrix $L_{q,n}$ and consequently $P_{q,n}$ depend on p, the characteristic of $\mathbb{F}_q$, and n. Since the finite field is usually fixed during our analysis we drop the symbol q and show the matrices as L_n and P_n for the sake of simplicity. In the next sections we see how special factorizations of P_n and L_n result in fast methods for the multiplication of these matrices by vectors.

4.5 Factorizations of the Conversion Matrices

The costs of computing the isomorphisms π_n and ν_n of Section 4.3 depend on the structure of the corresponding matrices. As in the last section, it is easier to initially study the structure of L_n and use this information to analyze P_n. The former study will be simplified by assuming n to be a power of p, say p^r, and extending the results to general n later. This simplification enables a recursive study of L_{p^r} which is shown in Example 10 and will be discussed in Lemma 15. This recursive structure is then later used in Theorem 17 to find a factorization of L_{p^r} into sparse matrices.



$$\left[\begin{array}{ccc|ccc|ccc}
1 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & 1\\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 2 & 0\\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 2\\
\hline
0 & 0 & 0 & 1 & 0 & 2 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1\\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0\\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right]$$

Figure 4.5: The block representation of the matrix L_9

Example 10. *The matrix L_9 has been computed in Example 7. The entries of this matrix are rewritten in Figure 4.5. In this figure L_9 is divided into nine blocks which have three rows and three columns each. These blocks can be grouped in three different groups. The ones which are colored in light gray contain only zero entries. We show these blocks as $Z_{3\times 3}$. The second group are the ones in blue and have structures which are very similar to the block in the first row and first column which is obviously L_3. Each block of this group in the ith row and jth column is the product of $\binom{j}{(j-i)/2}$ by L_3. The elements of the third group are colored in green. They are equal in our special example but if we represent the block in the first row and second column with L_3', the block in the ith row and jth column can be written as the product of $\binom{j}{(j-i-1)/2}$ and L_3'. Indeed the matrix L_3' can also be written as the product of the matrix Θ_3 which is*

$$\Theta_3 = \begin{pmatrix} 0 & 0 & 0\\ 0 & 0 & 1\\ 0 & 1 & 0 \end{pmatrix}$$

and L_3. Also the matrix L_9 can be written using the block representation:

$$L_9 = \begin{pmatrix} \binom{0}{0}L_3 & \binom{1}{0}\Theta_3 L_3 & \binom{2}{1}L_3\\ Z_{3\times 3} & \binom{1}{0}L_3 & \binom{2}{0}\Theta_3 L_3\\ Z_{3\times 3} & Z_{3\times 3} & \binom{2}{0}L_3 \end{pmatrix}$$

The above recursive relation is generally true between L_{p^r} and $L_{p^{r-1}}$ as will be proved in Lemma 15. To formally describe the above relation we define three matrices of *reflection, shifting*, and *factorization* denoted by Θ_n, Ψ_n, and B_r, respectively.

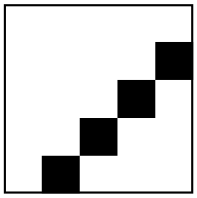


Figure 4.6: The matrix Θ_5

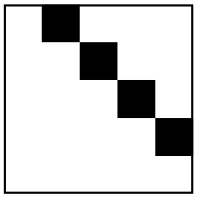


Figure 4.7: The matrix Ψ_5

Definition 11. *The entries of the* reflection *matrix* $\Theta_n = (\theta_{i,j})_{0\le i,j<n} \in \mathbb{F}_p^{n\times n}$ *are defined by the relation:*

$$\theta_{i,j} = \begin{cases} 1 & \textit{if } i+j=n, \\ 0 & \textit{otherwise}. \end{cases}$$

An example, Θ_5, is shown in Figure 4.6 where the coefficients equal to 0 and 1 are represented by empty and filled boxes, respectively. Left multiplication by Θ_n reflects a matrix horizontally and shifts the result by one row downwards.

Definition 12. *The entries of the* shifting *matrix* $\Psi_n = (\psi_{i,j})_{0\le i,j<n} \in \mathbb{F}_p^{n\times n}$ *are defined by the relation:*

$$\psi_{i,j} = \begin{cases} 1 & \textit{if } j-i=1, \\ 0 & \textit{otherwise}. \end{cases}$$

Right multiplication by Ψ_n shifts a matrix by one position upwards. As an example Ψ_5 is shown in Figure 4.7.

Definition 13. *Let $I_{p^{r-1}}$ be the identity $p^{r-1}\times p^{r-1}$ matrix and $\Theta_{p^{r-1}}$ and Ψ_p as in Definitions 11 and 12, respectively. Then we define the* factorization *matrix $B_r \in \mathbb{F}_p^{p^r\times p^r}$ to* *be:*

$$B_r = L_p \otimes I_{p^{r-1}} + (\Psi_p L_p) \otimes \Theta_{p^{r-1}},$$

in which $\otimes$ is the Kronecker or tensor product operator.

The following theorem gives us more information about the structure of B_r which can be helpful for constructing this matrix.

Theorem 14. *The matrix B_r can be split into $p \times p$ blocks $B^{(i_1,j_1)} \in \mathbb{F}_p^{p^{r-1}\times p^{r-1}}$ such that $B_r = (B^{(i_1,j_1)})_{0\le i_1,j_1<p}$ and*

$$B^{(i_1,j_1)} = \begin{cases} \textit{the zero block} & \textit{if } i_1 > j_1, \\ \binom{j_1}{(j_1-i_1)/2} I_{p^{r-1}} & \textit{if } i_1 \le j_1 \textit{ and } j_1 - i_1 \textit{ is even, and} \\ \binom{j_1}{(j_1-i_1-1)/2} \Theta_{p^{r-1}} & \textit{otherwise.} \end{cases}$$

Proof. For $0 \le i_0, j_0 < p^{r-1}$ we consider $(B^{(i_1,j_1)})_{i_0,j_0}$. Definition 13 implies that:

$$(B^{(i_1,j_1)})_{i_0,j_0} = (B_r)_{i_1 p^{r-1}+i_0, j_1 p^{r-1}+j_0} = (L_p)_{i_1,j_1}(I_{p^{r-1}})_{i_0,j_0} + (\Psi_p L_p)_{i_1,j_1}(\Theta_{p^{r-1}})_{i_0,j_0}.$$

Using Definition 12 the only nonzero entry of the i_1th row of Ψ_p is a 1 in the $i_1 + 1$st column, if $i_1 + 1 < p$, and hence $(\Psi_p L_p)_{i_1,j_1} = l_{i_1+1,j_1}$ and the above equation can be written as:

$$(B^{(i_1,j_1)})_{i_0,j_0} = l_{i_1,j_1}(I_{p^{r-1}})_{i_0,j_0} + l_{i_1+1,j_1}(\Theta_{p^{r-1}})_{i_0,j_0}. \tag{4.11}$$

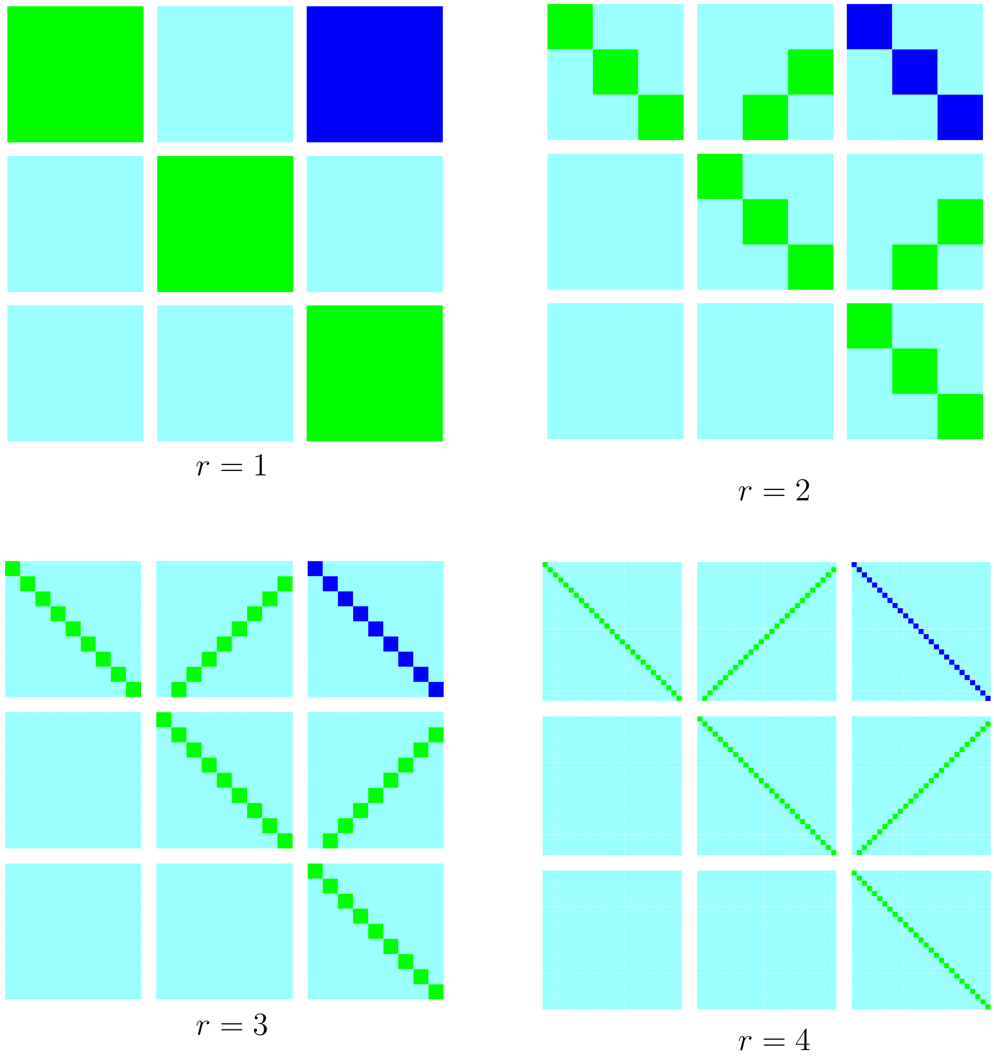
Now using Theorem 8:

- If $i_1 > j_1$, then l_{i_1,j_1} and l_{i_1+1,j_1} and hence also $(B^{(i_1,j_1)})_{i_0,j_0}$ are zero.

- If $i_1 \le j_1$ and $j_1 - i_1$ is even, then $l_{i_1,j_1} = \binom{j_1}{(j_1-i_1)/2}$ and l_{i_1+1,j_1} is zero.

- If $i_1 \le j_1$ and $j_1 - i_1$ is odd, then $l_{i_1,j_1} = 0$ and l_{i_1+1,j_1} equals $\binom{j_1}{(j_1-i_1-1)/2}$, since $j_1 - i_1 - 1$ is even.

□



$r = 1$

$r = 2$

$r = 3$

$r = 4$

Figure 4.8: The matrices B_r for $p = 3$ and 4 values of r

The matrices B_r for 4 values of $r = 1, 2, 3, 4$ are shown in Figure 4.8 with colors light blue, green, and dark blue for values of 0, 1, and 2 respectively. We now prove the following lemma.

Lemma 15. *The following equation holds for $r \geq 1$:*

$$L_{p^r} = B_r(I_p \otimes L_{p^{r-1}}). \tag{4.12}$$

Proof. For $0 \leq i, j < p^r$ we compute $(L_{p^r})_{i,j}$ by writing:

$$i = i_1 p^{r-1} + i_0, \quad j = j_1 p^{r-1} + j_0, \tag{4.13}$$

with $0 \leq i_1, j_1 < p$ and $0 \leq i_0, j_0 < p^{r-1}$. Since $p \cdot x = 0$:

$$\begin{aligned}&(x + x^{-1})^j = (x + x^{-1})^{j_1 p^{r-1}} (x + x^{-1})^{j_0} = (x^{p^{r-1}} + x^{-p^{r-1}})^{j_1} (x + x^{-1})^{j_0} = \\ &(\sum_{k_1 \in \mathbb{Z}} l_{k_1,j_1} x^{k_1 p^{r-1}})(\sum_{k_0 \in \mathbb{Z}} l_{k_0,j_0} x^{k_0}) = \sum_{k_0,k_1 \in \mathbb{Z}} l_{k_1,j_1} l_{k_0,j_0} x^{k_1 p^{r-1} + k_0}\end{aligned} \tag{4.14}$$

where $l_{k,j}$ is as Definition 6 and is zero for $|k| > |j|$. For the coefficient of $x^i = x^{i_1 p^{r-1} + i_0}$, which is $(L_{p^r})_{i,j}$, we have:

$$\begin{aligned}&k_1 p^{r-1} + k_0 = i_1 p^{r-1} + i_0 \Longrightarrow k_0 \equiv i_0 \mod p^{r-1} \Longrightarrow \\ &\begin{cases} k_0 = i_0 + t p^{r-1} \\ k_1 = i_1 - t \end{cases}, \text{with } t \in \mathbb{Z}.\end{aligned} \tag{4.15}$$

In the above equation except for $t = -1, 0$ we have $|i_0 + t p^{r-1}| \geq |p^{r-1}| > |j_0|$ which means $l_{i_0 + t p^{r-1}, j_0} = 0$ and hence:

$$(L_{p^r})_{i,j} = l_{i_1,j_1} l_{i_0,j_0} + l_{i_1+1,j_1} l_{i_0 - p^{r-1}, j_0} \tag{4.16}$$

in which $l_{i_1,j_1} = (L_p)_{i_1,j_1}$, $l_{i_0,j_0} = (L_{p^{r-1}})_{i_0,j_0}$, and we have seen in the proof of Theorem 14 that $l_{i_1+j_1} = (\Psi_p L_p)_{i_1,j_1}$. The value of $l_{i_0 - p^{r-1}, j_0}$ can be replaced by $l_{p^{r-1} - i_0, j_0}$ because of the symmetry of the binomial coefficients. The latter can again be replaced by $(\Theta_{p^{r-1}} L_{p^{r-1}})_{i_0,j_0}$ since for $0 < i_0 < p^{r-1}$ the only nonzero entry in the i_0th row of $\Theta_{p^{r-1}}$

is in the $p^{r-1} - i_0$th column and hence $(\Theta_{p^{r-1}} L_{p^{r-1}})_{i_0,j_0}$ is the entry in the $p^{r-1} - i_0$th row and j_0th column of $L_{p^{r-1}}$. For $i_0 = 0$ the entry $(\Theta_{p^{r-1}} L_{p^{r-1}})_{i_0,j_0}$ is zero since there is no nonzero entry in the i_0th row of $\Theta_{p^{r-1}}$, and l_{p^{r-1},j_0} is also zero since $j_0 < p^{r-1}$. Substituting all of these into (4.16) we will have the following equation:

$$(L_{p^r})_{i,j} = (L_p)_{i_1,j_1}(L_{p^{r-1}})_{i_0,j_0} + (\Psi_p L_p)_{i_1,j_1}(\Theta_{p^{r-1}} L_{p^{r-1}})_{i_0,j_0} \tag{4.17}$$

which together with (4.13) shows that:

$$L_{p^r} = L_p \otimes L_{p^{r-1}} + (\Psi_p L_p) \otimes (\Theta_{p^{r-1}} L_{p^{r-1}}). \tag{4.18}$$

It is straightforward, using Definition 13 to show that (4.18) is equivalent to (4.12). □

Example 16. *The matrix L_{81} is shown in Figure 4.9 where the numbers 0, 1, and 2 are shown with colors light blue, green, and dark blue respectively. The relation between L_{3^r} and $L_{3^{r-1}}$ is also shown in Figure 4.10.*

This recursive relation resembles that for the DFT matrix in Chapter 1 of Loan (1992) and enables us to find a matrix factorization for L_{p^r} in Theorem 17. Using this factorization the map of a vector under the isomorphism ν_n can be computed using $O(n \log n)$ operations as will be shown later in Section 4.6.

Theorem 17. *The matrix L_{p^r} can be written as:*

$$L_{p^r} = (I_1 \otimes B_r)(I_p \otimes B_{r-1}) \cdots (I_{p^{r-2}} \otimes B_2)(I_{p^{r-1}} \otimes B_1). \tag{4.19}$$

Proof. We use induction on r. If $r = 1$, then Θ_1 is zero and Definition 13 implies that:

$$L_p = B_1 = I_1 \otimes B_1.$$

Now assume that (4.19) is correct. Then using Lemma 15 :

$$\begin{aligned}
&L_{p^{r+1}} = B_{r+1}(I_p \otimes L_{p^r}) = \\
&B_{r+1} \cdot (I_p \otimes ((I_1 \otimes B_r)(I_p \otimes B_{r-1}) \cdots (I_{p^{r-2}} \otimes B_2)(I_{p^{r-1}} \otimes B_1))) = \\
&(I_1 \otimes B_{r+1}) \cdot (I_p \otimes B_r) \cdots (I_{p^{r-1}} \otimes B_2)(I_{p^r} \otimes B_1).
\end{aligned} \tag{4.20}$$

□

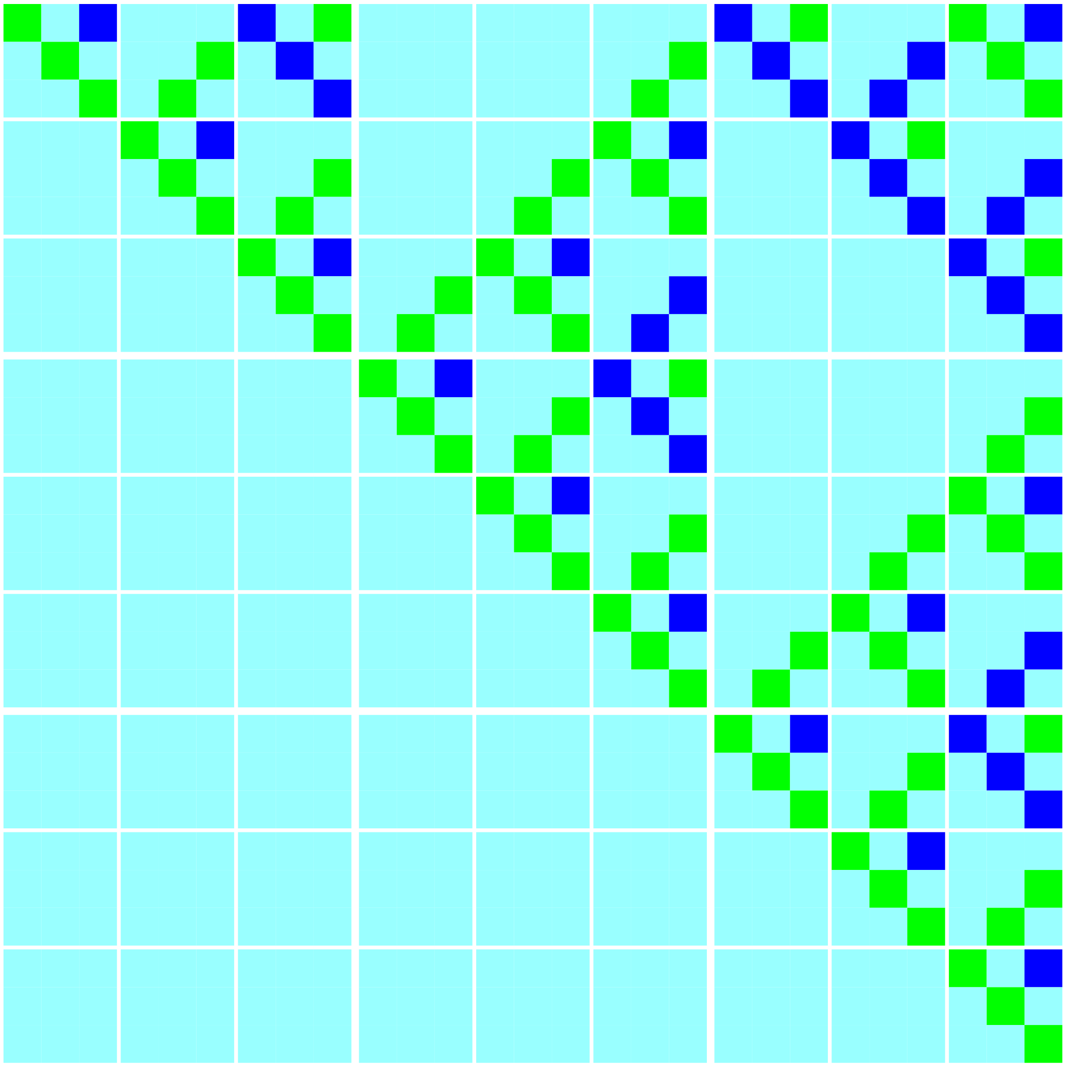


Figure 4.9: The matrix L_{81}

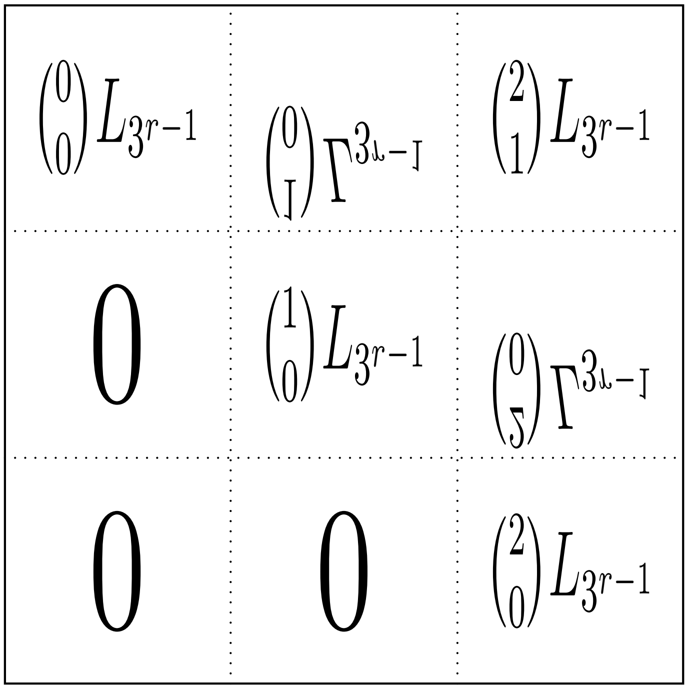

$\binom{0}{0}L_{3^{r-1}}$
$\binom{0}{1}L_{3^{r-1}}$
$\binom{2}{1}L_{3^{r-1}}$
0
$\binom{1}{0}L_{3^{r-1}}$
$\binom{0}{2}L_{3^{r-1}}$
0
0
$\binom{2}{0}L_{3^{r-1}}$

Figure 4.10: The relation between the matrix L_{3^r} and its sub-blocks. The sub-block at the ith row and jth column, if $i < j$ and $j - i$ is odd, is $\binom{j}{(j-i-1)/2}$ multiplied by the mirror of $L_{3^{r-1}}$.

Instead of multiplying L_{p^r} by a vector, we successively multiply the matrices in the factorization of (4.19) by that vector. In the next section we count the number of operations required for the computations of the mappings π_n and ν_n, but before that we informally describe the relation between Lemma 15 and the Pascal triangle. This informal description helps in better understanding of that lemma and can probably give some insights into data structures which are based on the modular Pascal triangle.

Consider a new triangle which is generated from the Pascal triangle in the following way (See Figures 4.11 and 4.12): At first a zero is inserted between any two horizontally adjacent entries of the Pascal triangle and every entry is reduced modulo p. This will result in the expansion of the Pascal triangle and the new triangle is then rotated 90 degrees counter-clockwise. This triangle can be split into two partitions as shown in Figure 4.11. In this figure the lower partition consists of the nonzero entries of L_{p^r}, whereas the upper partition contains the coefficients of the negative powers of x in the expansions of $(x + x^{-1})^j$. These negative powers construct, in a similar way to the definition of L_{p^r}, a new matrix which is shown by L' in Figure 4.11. The symmetry in the Pascal triangle can now be interpreted as the relation:

$$L'_{p^r} = \Theta \cdot L_{p^r},$$

and is demonstrated in the following example.

Example 18. *The powers $(x + x^{-1})^j \in \mathbb{F}_9[x]$, for $0 \leq j < 9$, were shown in Example 7 and can be used to construct L'_9. This matrix together with L_9 are shown in Figure 4.12-a. The entries $l_{i,j}$, for $i < j$, and odd $j - i$, and $l'_{i,j}$, for $j > p^r - i$ and even $j - i$, are zero, independent of the binomial coefficients, and are shown in gray while other entries are in black. The rotated Pascal triangle modulo 3 is shown in Figure 4.12-b for the ease of comparison.*

To analyze the recursive dependency between L_{p^r} and $L_{p^{r-1}}$ we write $0 \leq i, j < p^r$

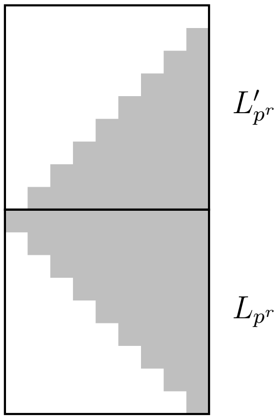
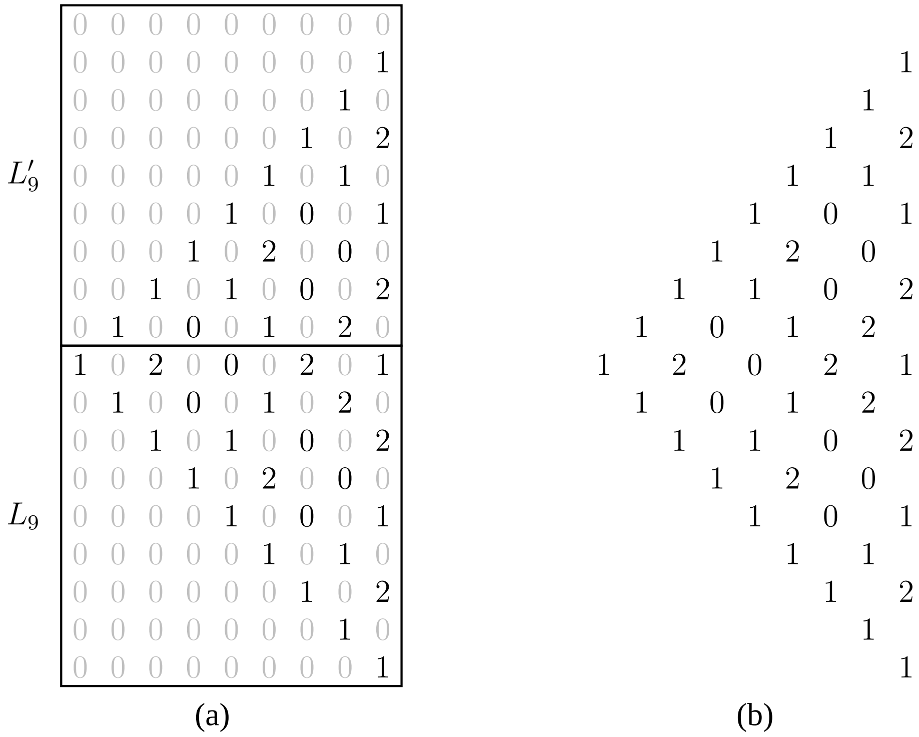


Figure 4.11: The relation between the matrices L_{p^r}, L'_{p^r}, and the Pascal triangle. The gray area is the Pascal triangle rotated 90 degrees counter-clockwise in which each entry is reduced modulo p, and a zero is inserted between any two horizontally adjacent entries.



$$\begin{array}{c|ccccccccc|}
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 \\
L'_9 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 1 & 0 & 2 & 0 & 0 & 0 \\
 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 2 \\
 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 2 & 0 \\
\hline
 & 1 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & 1 \\
 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 2 & 0 \\
 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 2 \\
 & 0 & 0 & 0 & 1 & 0 & 2 & 0 & 0 & 0 \\
L_9 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

(a)

(b)

Figure 4.12: (a) The entries of the matrices L_9 and L'_9 and (b) the rotated Pascal triangle modulo 3

as $i = i_1 p^{r-1} + i_0, j = j_1 p^{r-1} + j_0$ and expand:

$$\begin{aligned}(x+x^{-1})^j = (x+x^{-1})^{j_1 p^{r-1}}(x+x^{-1})^{j_0} = \\ \underbrace{(x^{p^{r-1}} + x^{-p^{r-1}})^{j_1}}_{\text{displacements}} \underbrace{(x+x^{-1})^{j_0}}_{\text{blocks}}.\end{aligned} \tag{4.21}$$

Since $0 \leq j_0 < p^{r-1}$ the coefficients of the powers of x in "blocks" make the concatenations of the columns of $L_{p^{r-1}}$ and $L'_{p^{r-1}}$ as shown in Figure 4.11 and Example 18. The terms in each block created by "blocks" are multiplied by one of the terms in "displacements" which are generally of the form $c_{j_1'} x^{j_1' p^{r-1}}$. This can be thought of as multiplying the block by the scalar $c_{j_1'}$ and moving it by $j_1' p^{r-1}$ positions downwards, in the matrix L_{p^r}. Different values of j_1 correspond to horizontal positions of blocks. Since j_1 is multiplied by p^{r-1} and the difference of two powers of x with nonzero coefficients in "displacements" is at least $2p^{r-1}$ and regarding the size of each block, $(2^{p^{r-1}} - 1) \times p^{r-1}$, the blocks are non-overlapping. This is shown in Figure 4.13-a. In this figure the blocks of non-negative and negative powers of x are shown with blue and green triangles, respectively. Note that although the triangles of each group have the same color, their entries are not equal. All of them are scalar multipliers of the same block.

Since the coefficients of negative powers of x are not directly present in L_{p^r} their corresponding blocks will be created by multiplying $\Theta_{p^{r-1}}$ by $L_{p^{r-1}}$. Now the two parts of B_r, i.e., $L_p \otimes I_{p^{r-1}}$ and $(\Psi_p L_p) \otimes \Theta_{p^{r-1}}$, can be considered as two masks which multiply the non-negative and negative blocks, $L_{p^{r-1}}$ and $L'_{p^{r-1}}$, by appropriate binomial coefficients and put them in the correct positions as shown in Figures 4.13-b and 4.13-c.

4.6 Costs of Computing ν_n and π_n

Multiplication by L_{p^r} consists of several multiplications by B_k for different values of k. Hence it is better to start the study by counting the required operations for multiplying B_k by a vector in $\mathbb{F}_q^{p^k}$.

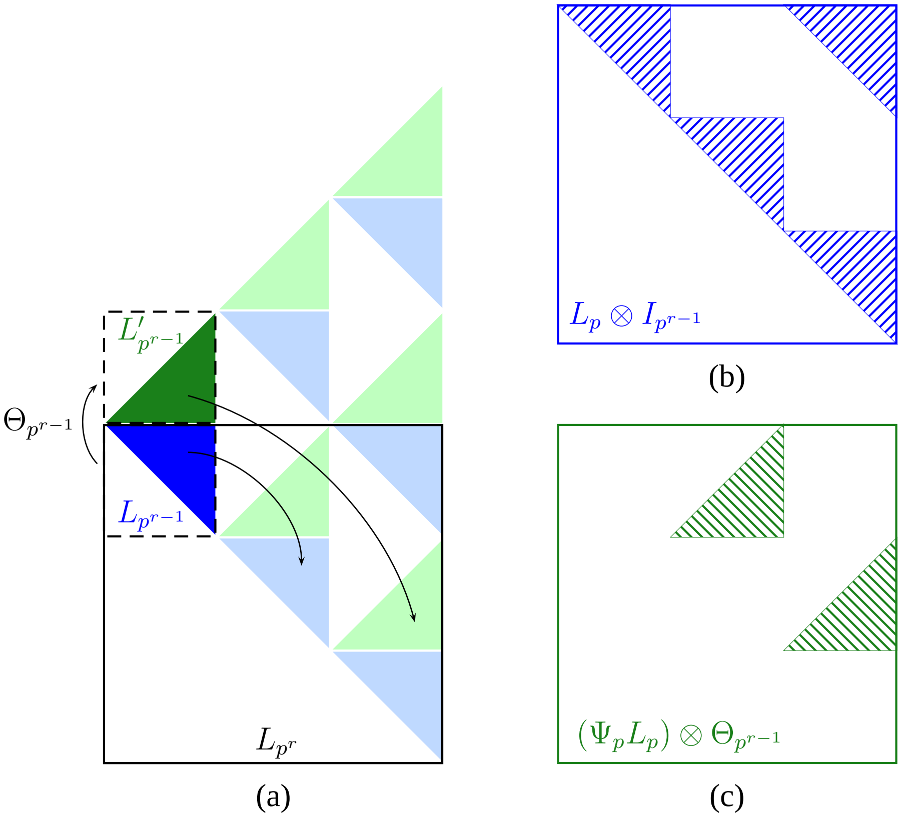

$L'_{p^{r-1}}$
$\Theta_{p^{r-1}}$
$L_{p^{r-1}}$
L_{p^r}
(a)
$L_p \otimes I_{p^{r-1}}$
(b)
$(\Psi_p L_p) \otimes \Theta_{p^{r-1}}$
(c)

Figure 4.13: (a) The recursive structure of the modified Pascal triangle together with the masking effect of B_r for (b) non-negative and (c) negative powers of x in the recursive construction of L_{p^r} in Lemma 15.

Definition 19. *Let B_k, for the finite field $\mathbb{F}_q$, be as in Definition 13 and p be the characteristic of $\mathbb{F}_q$. We define $\mu_{add}(k)$ and $\mu_{mult}(k)$ to be the number of additions and multiplications in $\mathbb{F}_q$ to multiply B_k by a vector in $\mathbb{F}_q^{p^k}$, respectively.*

It should be noted that to compute the functions $\mu_{add}(k)$ and $\mu_{mult}(k)$ we use the structure of the matrix B_k which is already known and hence the cost of adding an entry which is known to be zero to an element or that of multiplying one by an element is zero. As an example since B_1, for $p = 2$, is the identity matrix both $\mu_{add(1)}$ and $\mu_{mult(1)}$ are zero.

Lemma 20. *Let $\delta_{i,j}$ be the Kronecker delta, i.e., for $i, j \in \mathbb{N}$, $\delta_{i,j}$ is 1 if $i = j$ and otherwise 0. Then for $k \geq 1$ the function $\mu_{add}(k)$ is given by:*

$$\mu_{add}(k) = (p-1)(2p^k - p - 1)/4 - \delta_{p,2}/4.$$

Furthermore $\mu_{mult}(k) \leq (1 - \delta_{p,2})\mu_{add}(k)$.

$\binom{0}{0}I_{p^{k-1}}$	$\cdots$	$\binom{p-3}{(p-3)/2}\cdot$ $I_{p^{k-1}}$	$\binom{p-2}{(p-3)/2}\cdot$ $\Theta_{p^{k-1}}$	$\binom{p-1}{(p-1)/2}\cdot$ $I_{p^{k-1}}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
0	$\cdots$	$\binom{p-3}{0}\cdot$ $I_{p^{k-1}}$	$\binom{p-2}{0}\cdot$ $\Theta_{p^{k-1}}$	$\binom{p-1}{1}\cdot$ $I_{p^{k-1}}$
0	$\cdots$	0	$\binom{p-2}{0}\cdot$ $I_{p^{k-1}}$	$\binom{p-1}{0}\cdot$ $\Theta_{p^{k-1}}$
0	$\cdots$	0	0	$\binom{p-1}{0}\cdot$ $I_{p^{k-1}}$

(a)

$\binom{0}{0}I_{2^{k-1}}$	$\binom{1}{0}\Theta_{2^{k-1}}$
0	$\binom{1}{0}I_{2^{k-1}}$

(b)

Figure 4.14: The partitioning of B_k, according to Theorem 14, for two different cases of (a) odd prime p and (b) $p = 2$.

Proof. The block partitioning of B_k, according to Theorem 14, for two different cases of odd prime p and $p = 2$ are shown in Figure 4.14. As it can be seen the blocks on the main diagonal are of the form $\binom{j}{0} I_{p^{k-1}}$, for $0 \leq j < p$, which equals $I_{p^{k-1}}$. Hence all of the entries on the main diagonal of B_k are 1. If we denote the number of nonzero entries in the ith row of B_k by $H_i(B_k)$ then $H_i(B_k) > 0$ and the number of additions to multiply the ith row of B_k by a vector is at most $H_i(B_k) - 1$. This implies that the number of additions to multiply B_k by a vector is at most

$$\sum_{i=0}^{p^k-1} H_i(B_k) - p^k.$$

If we show the number of nonzero entries in B_k, or $\sum_{i=0}^{p^k-1} H_i(B_k)$, by $H(B_k)$, then the number of additions to multiply B_k by a vector can be written as:

$$\mu_{add}(k) = H(B_k) - p^k. \tag{4.22}$$

To compute $H(B_k)$ we use the fact that the nonzero blocks of B_k are scalar multiples of $I_{p^{k-1}}$ with p^{k-1} nonzero entries and $\Theta_{p^{k-1}}$ with $p^{k-1} - 1$ nonzero entries and we count the number of each of these blocks in B_k.

If p is odd there are $1 + 1 + \cdots + (p-1)/2 + (p-1)/2 + (p+1)/2 = 2\sum_{i=1}^{(p-1)/2} i + (p+1)/2$ blocks which are multiples of $I_{p^{k-1}}$ and $1 + 1 + \cdots + (p-1)/2 + (p-1)/2 = 2\sum_{i=1}^{(p-1)/2} i$ blocks which are multiples of $\Theta_{p^{k-1}}$. Since $\sum_{i=1}^{(p-1)/2} i = (p^2-1)/8$ we have:

$$\begin{aligned} H(B_k) - p^k =& \frac{p^2-1}{4}(p^{k-1} + p^{k-1} - 1) + \frac{p+1}{2}p^{k-1} - p^k = \\ & \frac{p-1}{4}(2p^k + 2p^{k-1} - p - 1) + \frac{-p^k + p^{k-1}}{2} = \\ & \frac{p-1}{4}(2p^k + 2p^{k-1} - p - 1 - 2p^{k-1}) = (p-1)(2p^k - p - 1)/4. \end{aligned} \tag{4.23}$$

For $p = 2$ the results of (4.23) is $2^{k-1} - 3/4$. In this case there are two blocks which are $I_{2^{k-1}}$ and one $\Theta_{2^{k-1}}$ in B_k. Hence $H(B_k) - 2^k = 2^{k-1} - 1 = 2^{k-1} - 3/4 - 1/4$.

We observe that $H(B_k) - p^k$ is also an upper bound for the number of multiplications in $\mathbb{F}_q$ since from the nonzero entries in B_k there are p^k entries which are on the main diagonal and are 1. These elements do not contribute to any multiplications. There are possibly other elements in B_k which are 1 but specifying them is complicated. If $p = 2$ there are only 1s and 0s in B_k and hence multiplication of B_k by a vector is done without any $\mathbb{F}_q$-multiplications. □

Using Lemma 20 we are now in the position to compute the cost of multiplication by L_{p^r} as shown in the following theorem.

Lemma 21. *Multiplying L_{p^r} by a vector in $\mathbb{F}_q^{p^r}$ for $r \geq 1$ requires $\eta(r)$ number of additions, where*

$$\eta(r) = r(p-1)p^r/2 - (p+1)(p^r-1)/4 - \delta_{p,2}(p^r-1)/(4(p-1)).$$

The number of multiplications is not larger than the number of additions.

Proof. It is clear from (4.20) that the number of additions and multiplications are

$$\sum_{k=1}^{r} p^{r-k}\mu_{\text{add}}(k) \text{ and } \sum_{k=1}^{r} p^{r-k}\mu_{\text{mult}}(k)$$

respectively and since $\mu_{mult}(r) \leq \mu_{add}(r)$ the total number of multiplications is not larger than the number of additions. Replacing $\mu_{\text{add}}(k)$ with its value from Lemma 20 we have:

$$\begin{aligned}&\sum_{k=1}^{r} p^{r-k}((p-1)(2p^k-p-1)-\delta_{p,2})/4 = \\ &\sum_{k=1}^{r}(p-1)p^r/2 - \frac{p^2-1+\delta_{p,2}}{4}\sum_{k=1}^{r} p^{r-k} = \\ &r(p-1)p^r/2 - \frac{p^2-1+\delta_{p,2}}{4}\sum_{k=1}^{r} p^{r-k}.\end{aligned} \tag{4.24}$$

Putting $\sum_{k=1}^{r} p^{r-k} = (p^r-1)/(p-1)$ in (4.24) gives the function $\eta(r)$ given above. □

The following theorem is the result of Lemma 21.

Theorem 22. *Multiplication of L_n from Definition 6 by a vector in $\mathbb{F}_q^n$ can be done using $O(n \log n)$ operations in $\mathbb{F}_q$.*

Proof. Let p be the characteristic of $\mathbb{F}_q$ and $r = \lceil \log_p n \rceil$. Obviously the above number of operations is upper bounded by the number of operations to multiply L_{p^r} by a vector in $\mathbb{F}_q^{p^r}$. This is given by the function $\eta(r)$ from Lemma 21. But we have $r - 1 < \log_p n \leq r$ and hence:

$$\eta(r) < rp^{r+1}/2 < p^2 n(\log_p n + 1)/2.$$

□

One interesting fact about this factorization, which distinguishes it from other recursive methods like FFT, is that it is not necessary to use all of the entries of L_{p^r} for values of n which are between p^{r-1} and p^r. To find a factorization of L_n in this case we use the factorization of L_{p^r}. Using (4.20) we can write:

$$L_{p^r} = A_0 A_1 \cdots A_{r-1},$$

where A_j, $0 \leq j < r$, are upper triangular and $A_j = I_{p^j} \otimes B_{r-j}$. Obviously L_n consists of the first n rows and columns of L_{p^r}. Now we can write:

$$L_n = A'_0 A'_1 \cdots A'_{r-1}, \tag{4.25}$$

where each A'_j is made up of the first n rows and columns of A_j because each of the involved matrices are upper triangular. This can be better explained by the following block matrix multiplication assuming that the sizes of the matrices are such that the operations are allowed.

$$\begin{pmatrix} A & B \\ 0 & C \end{pmatrix} \begin{pmatrix} D & E \\ 0 & F \end{pmatrix} = \begin{pmatrix} AD & AE + BF \\ 0 & CF \end{pmatrix}.$$

As it can be seen the first block of the product matrix depends only on the first blocks of the multiplicands.

In the next paragraphs we show that the cost of multiplying P_{p^r} by a vector can be computed by the same formulas as for the cost of multiplying L_{p^r} by a vector. First we observe that each B_r is nonsingular since it is upper-triangular and all of the entries on the main diagonal are 1. Now we can factorize P_{p^r}, since it is the inverse of L_{p^r}, using the factorization of L_{p^r} in (4.20):

$$P_{p^r} = (I_{p^{r-1}} \otimes B_1^{-1})(I_{p^{r-2}} \otimes B_2^{-1}) \cdots (I_p \otimes B_{r-1}^{-1})(I_1 \otimes B_r^{-1}). \tag{4.26}$$

Finding an exact expression for B_r^{-1} is not easy but the computation of an upper bound for the number of nonzero entries in this matrix is achieved by symbolically inverting B_r. As we will see later, the resulting matrix has a block representation in which each block is a polynomial in $\Theta_{p^{r-1}}$ with even or odd powers only. In the next paragraphs we count the number of nonzero entries in these blocks. The following lemma expresses the number of nonzero elements in the matrices constructed by such polynomials.

Definition 23. *We define even and odd polynomials to be polynomials of the forms $f(x^2)$ and $x \cdot f(x^2)$, for a general polynomial f, respectively. The product of two even or odd polynomials is an even polynomial whereas that of an even and an odd polynomial is an odd polynomial.*

Lemma 24. *Let $H = (h_{i,j})_{0 \le i,j < n} \in \mathbb{F}_p^{n \times n}$ be such that $H = g(\Theta_n)$ for a polynomial g. If $h_{i,j}$ is nonzero, then $i = j$ for even g and $i + j = n$ for an odd polynomial g. The number of nonzero entries in H is at most n and $n - 1$ for even and odd polynomial g, respectively.*

Proof. Let $\Phi_n \in \mathbb{F}_p^{n \times n}$ be the identity matrix with the top-left entry set to zero, i.e.,

$$(\Phi_n)_{i,j} = \begin{cases} 1 & \text{if } i = j \text{ and } i \neq 0, \\ 0 & \text{otherwise}. \end{cases}$$

We have $\Theta_n^2 = \Phi_n$ and $\Phi_n \Theta_n = \Theta_n$. It follows by induction that Θ_n^s, for $s > 0$, equals Φ_n and Θ_n for even and odd s, respectively. Hence sums of even and odd powers of Θ_n can have at most n and $n - 1$ nonzero entries, respectively. Note that $\Theta_n^0 = I_n$ is an even power of Θ which contains n nonzero entries. These nonzero entries must be on the positions where the entries of I_n and Θ_n are nonzero, respectively. □

Before we start the last theorem about the number of nonzero entries we need more information about the structure of B_k which is gathered in the following lemma.

Lemma 25. *Let $T = I_{p^k} - B_k = (T_{i,j})_{0\leq i,j<p}$ with $T_{i,j} \in \mathbb{F}_p^{p^{k-1}\times p^{k-1}}$. Then T has the following properties:*

1.
$$T_{i,j} = \begin{cases} \text{the zero block} & \text{if } i \geq j, \\ -\binom{j}{(j-i)/2} I_{p^{r-1}} & \text{if } j-i \text{ is even, and} \\ -\binom{j}{(j-i-1)/2} \Theta_{p^{r-1}} & \text{otherwise,} \end{cases}$$

2. *For any $s \geq 0$ the blocks of $T^s = (T_{i,j}^{(s)})_{0\leq i,j<p}$ with $T_{i,j}^{(s)} \in \mathbb{F}_p^{p^{k-1}\times p^{k-1}}$ satisfy*
$$T_{i,j}^{(s)} = \begin{cases} \text{the zero block} & \text{if } j-i < s, \\ g_{i,j}(\Theta_{p^{k-1}}) & \text{otherwise,} \end{cases} \tag{4.27}$$
where $g_{i,j} \in \mathbb{F}_p[x]$ is odd and even for $j-i$ odd and even, respectively, and

3. $T^p = 0$.

Proof. Part 1 can be directly verified by $T = I_{p^k} - B_k$ and Theorem 14. Since T is strictly upper triangular the blocks on the main diagonal of T^s and $s-1$ diagonals on top of that are zero, i.e., $T_{i,j}^{(s)}$ is the zero block whenever $j-i<s$. To show the condition on the polynomials $g_{i,j}$ we use again induction on s. For the beginning, I_{p^k} and T obviously satisfy (4.27) according to Part 1. Now assume that this equation is satisfied for all integers $s < s_0$ and let $s_1, s_2 < s_0$ and $s_0 = s_1 + s_2$. Then the block on the ith row and jth column of T^{s_0} is:
$$T_{i,j}^{(s_0)} = \sum_{t=1}^{p} T_{i,t}^{(s_1)} T_{t,j}^{(s_2)}. \tag{4.28}$$
Now if $j-i$ is even, $j-t$ and $t-i$ must be both even or odd. In these cases two even or odd polynomials of $\Theta_{p^{k-1}}$ are multiplied and the resulting polynomial will be even. If, on the other hand, $j-i$ is odd either $t-i$ or $j-t$ is odd and the other one is even. In this case two polynomials of $\Theta_{p^{k-1}}$ are multiplied, so that one of them is odd and the other one even. This results in an odd polynomial in $\Theta_{p^{k-1}}$. Part 3 is also a direct result of Part 2 since all of the blocks satisfy $j-i<p$. □

Lemma 26. *Multiplication of B_k^{-1} by a vector in $\mathbb{F}_q^{p^k}$ requires at most $\mu_{add}(k)$ and $\mu_{mult}(k)$ additions and multiplications in $\mathbb{F}_q$, respectively where $\mu_{add}(k)$ and $\mu_{mult}(k)$ are given in Lemma 20.*

Proof. Since $T^p = 0$ we can write:

$$I_{p^k} - T^p = I_{p^k} = (I_{p^k} - T)(I_{p^k} + T + \cdots + T^{p-1}).$$

Hence using the definition of T in Lemma 25:

$$I_{p^k} = B_k \cdot (I_{p^k} + T + \cdots + T^{p-1}) \Longrightarrow B_k^{-1} = I_{p^k} + T + \cdots + T^{p-1}. \qquad (4.29)$$

Lemma 25 shows that each T^s, for $s \geq 0$, and hence B_k^{-1} can be partitioned in a way similar to Lemma 25, such that the block on the ith row and jth column is the zero block for $i > j$ and an even or an odd polynomial in $\Theta_{p^{k-1}}$ for even and odd $j - i$, respectively. Note that the zero blocks in the identity matrix are even and odd polynomials in $\Theta_{p^{k-1}}$. These even and odd polynomials have at most n and $n-1$ nonzero coefficients, respectively according to Lemma 24. Now the same method as that of Lemma 20 shows that the number of $\mathbb{F}_q$-additions and multiplications are bounded by $\mu_{add}(k)$ and $\mu_{mult}(k)$, respectively. □

Theorem 27. *Multiplication of P_n from Definition 9 by a vector in $\mathbb{F}_q^n$ can be done using $O(n \log n)$ operations in $\mathbb{F}_q$.*

Proof. Lemma 26 and the same argumentation as Lemma 21 show that multiplication of P_{p^r} by a vector is done using $\eta(r)$ operations, where $\eta(r)$ is given in Lemma 21. Now the proof is similar to Theorem 22. □

We conclude this section with the following theorem. Although its result is not concerned with normal basis multiplication directly, it emphasizes the most important property of our multiplier. Namely a specific change of basis in $\mathbb{F}_{q^n}$ which can be done using $O(n \log n)$ instead of $O(n^2)$ operations, which is the cost of general basis conversion in $\mathbb{F}_{q^n}$.

Theorem 28. *Let $\mathcal{N}$ be a type-II normal basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ generated by the normal element $\beta + \beta^{-1}$ and*

$$\mathcal{P} = (1, \beta + \beta^{-1}, \cdots, (\beta + \beta^{-1})^{n-1})$$

be the polynomial basis generated by the minimal polynomial of $\beta+\beta^{-1}$. Then the change of representation between the two bases $\mathcal{N}$ and $\mathcal{P}$ can be done using $O(n \log n)$ operations in $\mathbb{F}_q$.

Proof. The $\mathcal{N}$-basis vector representation of an element is converted to the extended permuted representation, as in Figure 4.2, without any arithmetic operations. Then the matrix P_{n+1} is multiplied by this vector using at most $\eta(r)$ operations, where $r = \lceil \log_p n \rceil$ and p is the characteristic of $\mathbb{F}_q$, and the coefficient of $(\beta + \beta^{-1})^n$ is converted to the polynomial basis using at most $2n$ additions and multiplications in $\mathbb{F}_q$. This cost is $O(n \log n)$ according to Theorem 22.

To convert the representation of an element from $\mathcal{P}$ into $\mathcal{N}$ we insert a zero, as the coefficient of $(\beta+\beta^{-1})^n$, to the end of the representation vector in $\mathcal{P}$. Then L_{n+1} is multiplied by the resulting vector and finally the first entry which is the constant term is converted to the normal basis representation by multiplying it by the vector representation of 1 using at most $2n$ operations in $\mathbb{F}_q$. This again can be done using $O(n \log n)$ operations. □

4.7 Other Costs

There are two other operations in our multiplier which will be discussed in this section. Namely polynomial multiplication and conversion from the extended permuted representation to the normal basis representation.

The polynomial multiplication method can be selected arbitrarily among all available methods depending on the polynomial lengths and the implementation environments. Chapter 3 was devoted to moderate polynomial sizes which are applicable to cryptography. Although Table 3.6 of that chapter compares our multipliers with others for polynomial lengths up to 8192, the methods can be applied to larger polynomials as well. For a thorough analysis of other methods of polynomial multiplication see von zur Gathen & Gerhard (2003), Chapter 8. We assume the polynomial multipliers of Chapter 3 to require $\lceil 7.6\, n^{\log_2 3} \rceil$ two-input gates. The above expression has been computed as an upper bound for the area of those multipliers in the interval $160 < n < 10000$.

Another cost which we analyze is the number of bit operations to convert from extended permuted to the permuted representation. By multiplying the polynomials of length $n + 1$ the product which is of length $2n + 1$ is converted to a linear combination of $\beta^i + \beta^{-i}$ for $0 \leq i \leq 2n$. These values should be converted to the permuted representation, i.e., $\beta^i + \beta^{-i}$ for $1 \leq i \leq n$. This conversion is done using the fact that β is a $2n + 1$st root of unity. The costs for the case of odd prime numbers are given in the next theorem.

Theorem 29. *Let p, the characteristic of $\mathbb{F}_{q^n}$, be an odd prime number. Conversion from extended permuted representation of the product in Figure 4.2 into the permuted basis can be done using at most $2n$ additions and n scalar multiplications in $\mathbb{F}_q$.*

Proof. The conversion from the extended permuted representation to the permuted basis must be done for the constant term and $\beta^i + \beta^{-i}$ when $i > n$. Since β is a $2n + 1$th root of unity $\beta^{n+k} = \beta^{n+1-k}$ for $1 \leq k \leq n$ and $\beta^{n+k} + \beta^{-n-k} = \beta^{n+1-k} + \beta^{-n-1+k}$. Hence the corresponding coefficients must be added together. This is done using n additions. The mapping of the constant term is done by multiplying it with the vector of representation of 1 in the permuted normal basis. This is done with at most n additions and n multiplications in $\mathbb{F}_q$. □

The above task can be done using n additions when the characteristic of the finite field is 2 since in that case the constant term vanishes, as will be shown later using the following lemma.

Lemma 30. *For any positive integer n the binomial coefficient $\binom{2n}{n}$ is an even number.*

Proof. This can be easily proven using Lucas' theorem. This theorem (see PlanetMath (2002)) states that for any two positive integers a and b with p-adic representations

$a_{m-1}a_{m-2}\cdots a_0$ and $b_{m-1}b_{m-2}\cdots b_0$ respectively, we have:

$$\binom{a}{b} \equiv \binom{a_{m-1}}{b_{m-1}}\binom{a_{m-2}}{b_{m-2}}\cdots\binom{a_0}{b_0} \mod p. \tag{4.30}$$

Let $n_{m-1}n_{m-2}\cdots n_0$ be the binary representation of n and k be its first nonzero digit from the right, i.e., for each $j < k$ we have $n_j = 0$ and $n_k = 0$. Since the binary representation of $2n$ is that of n shifted by one position to left, the digit on the kth position of the binary representation of $2n$ is zero. The relation

$$\binom{2n}{n} \equiv 0 \mod 2$$

is hence the result of the fact that $\binom{0}{1}$ is equal to zero and (4.30). □

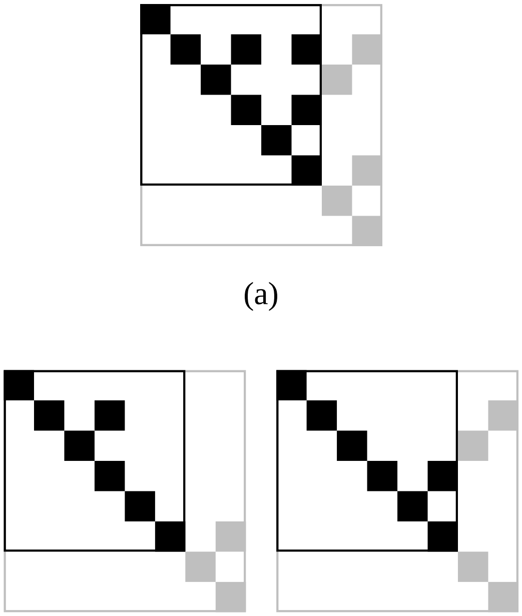
Theorem 31. *Let $\varphi_c(x)$ be the polynomial representation of the product c, as shown in Figure 4.2 and q be a power of 2. Then the constant term in $\varphi_c(x)$ is zero.*

Proof. According to Theorem 8 and Lemma 30 the entry $l_{0,0}$ is the only nonzero entry $l_{0,j}$ of L_{2^k} , for every integer k and $0 \leq j < 2^k$. On the other hand, as we saw in Section 4.4, zeros are inserted to the beginning of the permuted normal representations of a and b and the entries at the index 0 of these two new vectors are zero. Hence the constant terms in polynomials φ_a and φ_b in Figure 4.2 are zero and since φ_c is the product of φ_a and φ_b the constant term in that polynomial is zero, too. □

Using the materials which are presented herein we can summarize the costs of our multiplier in the following theorem. Since we can use any suitable polynomial multiplier, the presented costs depend on the polynomial multiplication methods used.

Theorem 32. *Let $\mathbb{F}_{q^n}$ be a finite field of characteristic p, which contains an optimal normal basis of type 2. Let further $\delta_{i,j}$ be the Kronecker delta as stated in Lemma 20, $\mathsf{M}(n)$ be the number of $\mathbb{F}_q$-operations to multiply two polynomials of degree $n-1$, $\eta(r)$ be as given in Lemma 21, $r_1 = \lceil \log_p(n+1)\rceil$, and $r_2 = \lceil \log_p(2n+1)\rceil$. Multiplication in this finite field, in normal basis, can be done using at most*

$$n + 2(1-\delta_{p,2})n + 2\eta(r_1) + \eta(r_2) + \mathsf{M}(n+1)$$



(a)

(b)

Figure 4.15: (a) The matrices P_6 and P_8 and (b) their factorizations. All nonzero entries which belong only to P_6 are in black and other nonzero entries in P_8 are in gray.

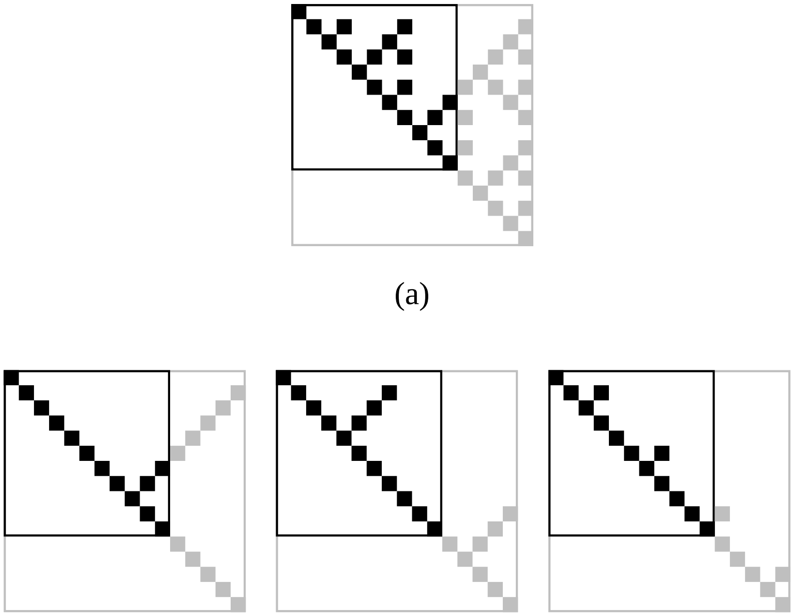
operations in $\mathbb{F}_q$. *For sufficiently large* n *the above expression is upper bounded by* $\mathsf{M}(n+1) + 3n + 2(2n+1)p^2 \log_p(2n+1)$.

It should be pointed out that for the case $p = 2$ we have $T^2 = 0$, for the matrix L in Theorem 26, and Equation 4.29 implies that each B_k is its own inverse and computing π_n has the same cost as ν_n.

The matrices L_{11} and P_6 when $p = 2$, i.e., the case of the example in Section 4.3 and their factorizations are shown in Figures 4.15 and 4.16, respectively.

4.8 Comparison

The multiplier which is proposed in this section is especially efficient when the extension degree n is much larger than the size of the ground field q. One practical application of this kind is the cryptography in fields of characteristic 2. In this section we compare



(a)

(b)

Figure 4.16: (a) The matrices L_{11} and L_{16} and (b) their factorizations. All nonzero entries which belong only to L_{16} are in gray, whereas common entries of L_{16} and L_{11} are in black.

this multiplier with some other structures, from the literature, which are proposed for multiplication in such fields using normal bases of type 2. The field extensions which are discussed here are from Table 4.1.

The first structure, which we study here is the circuit of Sunar & Koç (2001) with $n(5n-1)/2$ gates. The second circuit is from Gao *et al.* (1995). The idea behind this multiplier is to consider the representation

$$a_1(\beta+\beta^{-1})+\cdots+a_n(\beta^n+\beta^{-n})$$

as the sum of two polynomials

$$a_1\beta+\cdots+a_n\beta^n \text{ and } a_n\beta^{-n}+\cdots+a_1\beta^{-1}.$$

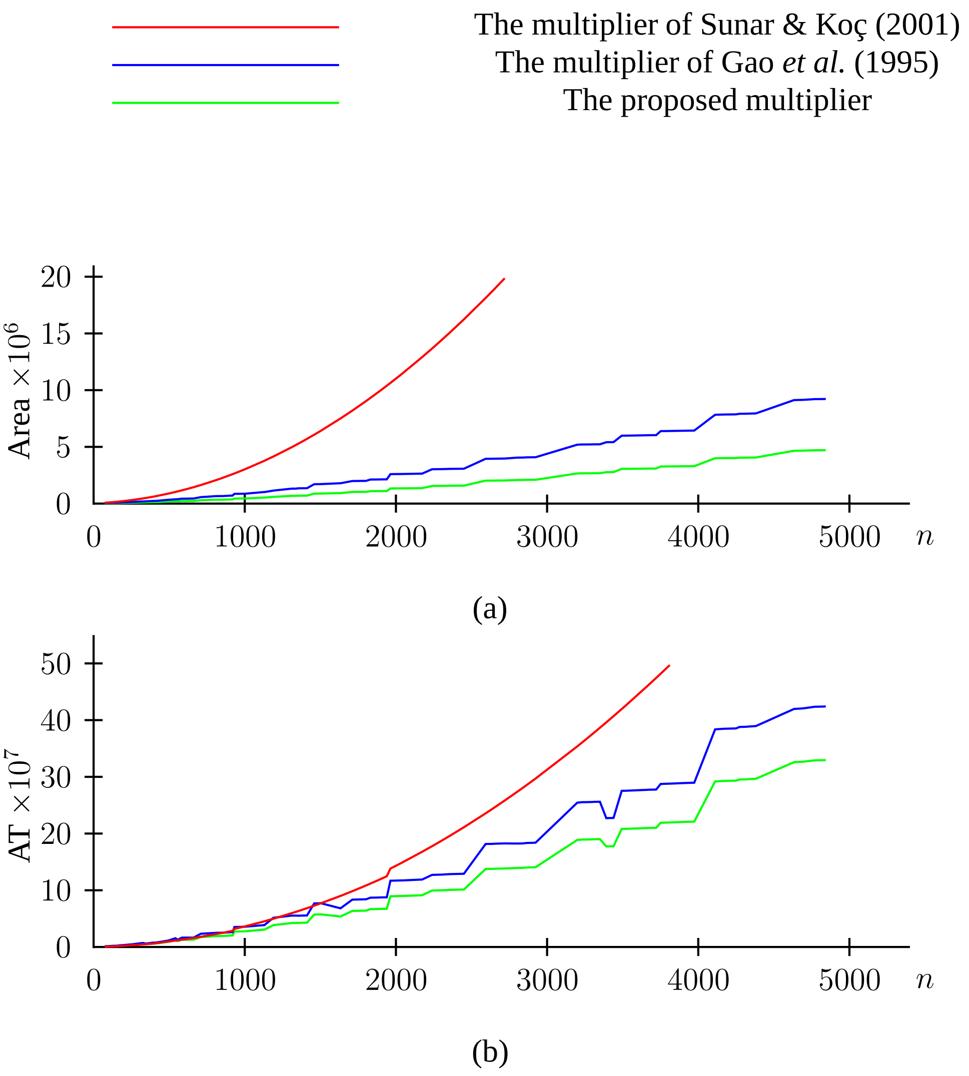
To multiply two elements four polynomials of degree n should be multiplied together. However, because of the symmetry only two multiplications are necessary which also result in the other two products by mirroring the coefficients. The cost of a multiplication using this circuit is $2\mathsf{M}(n) + 2n$, where $\mathsf{M}(n)$ is the cost of multiplying two polynomials of length n.

Since we are interested in hardware implementations of algorithms we compare the circuits with respect to both area and area-time. The propagation delay of the multiplier of Sunar & Koç (2001) is $1 + \lceil \log_2 n \rceil$ gates. The propagation delay of the multiplier of this chapter consists of two parts: the first one belongs to the conversion circuits which is $2+2\lceil \log_2 n \rceil$ and the other part corresponds to the polynomial multiplier. We compute the propagation delay of each polynomial multiplier for that special case. The propagation delay of the multiplier of Gao *et al.* (1995) is two plus the delay of each polynomial multiplier which must again be calculated for each special case.

The area and AT parameters of these three circuits are compared with each other and the results are shown in Figure 4.17. In these diagrams polynomial multiplication is done using the methods of Chapter 3. As it can be seen the area of the proposed multiplier is always better than the other two structures. But the AT parameter is larger for small finite fields. This shows that, as we have mentioned, this method is appropriate for applications where only small area is available or where the finite fields are large. Economical applications, where small FPGAs should be used are situations of this sort. The AT parameter of the proposed multiplier is $O(n \log^3 n (\log \log n)^3)$, whereas that of the structure in Sunar & Koç (2001) is $O(n^2 \log n)$.

4.9 Conclusion

This chapter presented a new method for multiplication in finite fields using optimal normal bases of type 2. The area of this multiplier is smaller than other proposed structures but has a higher propagation delay, hence is suitable for low area implementations. The most important property of this multiplier, which is inherited from its conceptual parent in Gao *et al.* (1995), is the ability of using polynomial multipliers for normal bases. This enables designers to select the most appropriate structure, from the well studied area



(a)

(b)

Figure 4.17: Comparing the (a) area (as the number of two-input gates) and (b) the AT parameter (as the product of the number of two-input gates and the delay of a single gate) of three multipliers for binary finite fields with extension degrees from Table 4.1.

of polynomial multipliers, to fit their special conditions. The advantage of this structure, compared to that of Gao *et al.* (1995), is the reduction of the number of operations from two polynomial multiplications to one multiplication plus a small circuit of size $O(n \log n)$ for the change of representation. The materials of this chapter were arranged in the following parts:

- First the definitions of the Gauss periods and optimal normal bases of type 2 reviewed from the literature.

- The structure of the multiplier and the definitions of the used data structures were presented in Section 4.3.

- The data structures for the change of representations were introduced. Some facts about their matrices were proved, which resulted in special factorizations. These factorizations allowed the change of representations to be done using $O(n \log n)$ operations.

- The costs corresponding to the other parts of the multiplier are briefly studied.

- Finally Section 4.8 compared the area and AT measures of the proposed multiplier with two other structures from the literature for the finite fields $\mathbb{F}_{2^n}$, for $160 < n < 5000$, in which optimal normal bases of type 2 exist. Results showed that the asymptotically small area of the multiplier makes it even attractive for elliptic curve cryptography, where the finite field sizes are not very large ($160 < n < 600$). But designers should note the long propagation delay and use it only for applications where the area is limited or too expensive or for large finite fields.

Chapter 5

Conclusion and Future Works

The aim of this work is to present the design stages of an elliptic curve co-processor. Elliptic curve cryptography is going to be an important part of cryptography because of its relatively short key length and higher efficiency as compared to other well-known public key crypto-systems like RSA. Chapter 1 contains a very brief overview of cryptography, FPGAs, and parameters which are used for designing the circuit.

Chapter 2 studies the stages of the design of a high performance elliptic curve co-processor. It is shown in this chapter that for small area applications, the combination of polynomial basis for the finite field representation and the Montgomery method for the point representation and scalar multiplication, is best. In addition, it is shown in this chapter that it is always better to use as much parallelism as possible in the finite field arithmetic level rather than in the bit-level. This means that for example, if allowed by the algorithm two serial multipliers are better than a single multiplier which produces two output bits in one clock cycle. A comparison between all of the published reports is not possible due to differences in hardware platforms. But the comparison with a circuit on the same FPGA shows the high performance of the co-processor presented here.

The rest of this work studies different methods to improve the efficiency of the finite field multiplication as a ground operation in elliptic curve cryptography. The results of Chapter 3 propose a novel pipelined architecture for the multiplication of polynomials over $\mathbb{F}_2$. This chapter begins with a machine-independent improvement of the Karatsuba method by combining different multiplication methods and continues with the application of pipelining as a machine-dependent optimization to further improve the results. Although these results are not built into the designed co-processor, the comparisons between this structure and other classical methods for 240-bit polynomials show the suitability for applications in elliptic curve cryptography by covering the NIST finite finite field $\mathbb{F}_{233}$.

Finally Chapter 4 presents a small area normal basis multiplier. This multiplier reduces the multiplication in optimal normal bases of type 2 to one polynomial multiplication and a small circuit of size $O(n \log n)$. These results are probably not directly applicable to generic elliptic curves because of the high propagation delay in their circuits and the fact that for these curves polynomial bases are better, as is shown in Chapter 2. They can instead be applied to Koblitz curves, where several squarings should be done for a single point multiplication (see Hankerson *et al.* (2003), Section 4.1.2, Page 163) or to finite field inversion, where there are a lot of squarings compared to multiplications. It is better to perform these squarings in normal bases. Another advantage of this structure is its efficiency for the change of representation between polynomial and normal bases. This change of representation, which is used by Park *et al.* (2003) to strengthen the systems against side channel attacks, can be done by the multiplier of this chapter using $O(n \log n)$ instead of the assumed $O(n^2)$ operations. Another possible application of this system is in systems where several normal basis multiplications can be done in parallel. In this case pipelining can be used to decrease the latency of the system while keeping the area to a minimum. This happens, in fields of characteristic 3 for identity based cryptosystems, as also mentioned in Granger *et al.* (2005). These can be considered as the future research directions of this project.

Appendix A

Karatsuba multiplication Formulas for Polynomials of Degrees 2 and 7 for fields of characteristic 2

A.1 Degree 2

$$\begin{array}{lll}
a(x) = a_2x^2 + a_1x + a_0, & & \\
b(x) = b_2x^2 + b_1x + b_0), & & \\
P_0 = a_0b_0, & P_1 = (a_0 + a_1)(b_0 + b_1), & P_2 = a_1b_1, \\
P_3 = (a_0 + a_2)(b_0 + b_2), & P_4 = (a_1 + a_2)(b_1 + b_2), & P_5 = a_2b_2,
\end{array}$$

$$\begin{aligned}
a(x)b(x) = \quad & P_0(1 + x + x^2) + P_1x + P_2(x + x^2 + x^3)+ \\
& P_3x^2 + P_4x^3 + P_5(x^2 + x^3 + x^4).
\end{aligned}$$

A.2 Degree 7

$$\begin{aligned}
&a(x) = (a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0),\\
&b(x) = (b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0),\\
&P_0 = a_0b_0, \quad P_1 = a_1b_1, \quad P_2 = a_2b_2, \quad P_3 = a_3b_3, \quad P_4 = a_4b_4, \quad P_5 = a_5b_5,\\
&P_6 = a_6b_6, \quad P_7 = a_7b_7,\\
&P_8 = (a_0 + a_1)(b_0 + b_1), \quad P_9 = (a_0 + a_2)(b_0 + b_2), \quad P_{10} = (a_1 + a_3)(b_1 + b_3),\\
&P_{11} = (a_2 + a_3)(b_2 + b_3), \quad P_{12} = (a_0 + a_4)(b_0 + b_4), \quad P_{13} = (a_1 + a_5)(b_1 + b_5),\\
&P_{14} = (a_2 + a_6)(b_2 + b_6), \quad P_{15} = (a_3 + a_7)(b_3 + b_7), \quad P_{16} = (a_4 + a_5)(b_4 + b_5),\\
&P_{17} = (a_4 + a_6)(b_4 + b_6), \quad P_{18} = (a_5 + a_7)(b_5 + b_7), \quad P_{19} = (a_6 + a_7)(b_6 + b_7),\\
&P_{20} = (a_0 + a_1 + a_2 + a_3)(b_0 + b_1 + b_2 + b_3), P_{21} = (a_0 + a_1 + a_4 + a_5)(b_0 + b_1 + b_4 + b_5),\\
&P_{22} = (a_0 + a_2 + a_4 + a_6)(b_0 + b_2 + b_4 + b_6), P_{23} = (a_1 + a_3 + a_5 + a_7)(b_1 + b_3 + b_5 + b_7),\\
&P_{24} = (a_1 + a_2 + a_5 + a_6)(b_1 + b_2 + b_5 + b_6), P_{25} = (a_2 + a_3 + a_6 + a_7)(b_2 + b_3 + b_6 + b_7),\\
&P_{26} = (a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7)(b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7),
\end{aligned}$$

$$\begin{aligned}
a(x)b(x) = \quad & P_0(1 + x^1 + x^2 + x^3 + x^4 + x^5 + x^6 + x^7)+\\
& P_8(x^1 + x^3 + x^5 + x^7) + P_1(x^1 + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8)+\\
& P_9(x^2 + x^3 + x^6 + x^7) + P_{20}(x^3 + x^7) + P_{10}(x^3 + x^4 + x^7 + x^8)+\\
& P_2(x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9) + P_{11}(x^3 + x^5 + x^7 + x^9)+\\
& P_3(x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10}) + P_{12}(x^4 + x^5 + x^6 + x^7)+\\
& P_{21}(x^5 + x^7) + P_{13}(x^5 + x^6 + x^7 + x^8) + P_{22}(x^6 + x^7) + P_{26}(x^7)+\\
& P_{23}(x^7 + x^8) + P_{14}(x^6 + x^7 + x^8 + x^9) + P_{24}(x^7 + x^9)+\\
& P_{15}(x^7 + x^8 + x^9 + x^{10}) + P_4(x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + x^{11})+\\
& P_{16}(x^5 + x^7 + x^9 + x^{11}) + P_5(x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{12})+\\
& P_{17}(x^6 + x^7 + x^{10} + x^{11}) + P_{25}(x^7 + x^{11}) + P_{18}(x^7 + x^8 + x^{11} + x^{12})+\\
& P_6(x^6 + x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{12} + x^{13})+\\
& +P_{19}(x^7 + x^9 + x^{11} + x^{13}) + P_7(x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{12} + x^{13} + x^{14}).
\end{aligned}$$

Bibliography

Y. ASANO, T. ITOH & S. TSUJII (1989). Generalised fast algorithm for computing multiplicative inverses in $GF(2^m)$. *Electronics Letters* **25**(10), 664–665.

DANIEL V. BAILEY & CHRISTOF PAAR (1998). Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms. In *Advances in Cryptology: Proceedings of CRYPTO '98,* Santa Barbara CA, HUGO KRAWCZYK, editor, number 1462 in Lecture Notes in Computer Science, 472–485. Springer-Verlag. ISBN 3-540-64892-5.

M. BEDNARA, M. DALDRUP, J. SHOKROLLAHI, J. TEICH & J. VON ZUR GATHEN (2002a). Reconfigurable Implementation of Elliptic Curve Crypto Algorithms. In *Proc. of The 9th Reconfigurable Architectures Workshop (RAW-02)*. Fort Lauderdale, Florida, U.S.A.

M. BEDNARA, M. DALDRUP, J. SHOKROLLAHI, J. TEICH & J. VON ZUR GATHEN (2002b). Tradeoff Analysis of FPGA Based Elliptic Curve Cryptography. In *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS-02)*, volume V, 797–800. Scottsdale, Arizona, U.S.A.

RICHARD E. BLAHUT (1985). *Fast Algorithms for Digital Signal Processing*. Addison-Wesley, Reading MA.

IAN BLAKE, GADIEL SEROUSSI & NIGEL SMART (1999). *Elliptic Curves in Cryptography*. Number 265 in London Mathematical Society Lecture Note Series. Cambridge University Press.

A. BRAUER (1939). On addition chains. *Bulletin of the American Mathematical Society* **45**, 736–739.

DAVID G. CANTOR (1989). On Arithmetical Algorithms over Finite Fields. *Journal of Combinatorial Theory, Series A* **50**, 285–300.

HENRI COHEN, ATSUKO MIYAJI & TAKATOSHI ONO (1998). Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In *Advances in Cryptology - ASIACRYPT'98*, K. OHTA & D. PEI, editors, number 1514 in Lecture Notes in Computer Science, 51–65. Springer-Verlag. ISBN 3-540-65109-8. ISSN 0302-9743.

MICHAEL DALDRUP (2002). *Entwurf eines FPGA-basierten Koprozessors zur Kryptographie mit Elliptischen Kurven*. Diplomarbeit, University of Paderborn.

WHITFIELD DIFFIE & MARTIN E. HELLMAN (1976). New directions in cryptography. *IEEE Transactions on Information Theory* **IT-22**(6), 644–654.

T. ELGAMAL (1985). A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory* **IT-31**(4), 469–472.

FIPS PUB 180-1 (1993). *Secure Hash Standard*. U.S. Department of Commerce / National Institute of Standards and Technology. Federal Information Processings Standards Publication 180-1.

FIPS PUB 186-2 (2000). *Digital Signature Standard (DSS)*. U.S. Department of Commerce / National Institute of Standards and Technology. Federal Information Processings Standards Publication 186-2.

LIJUN GAO, SARVESH SHRIVASTAVA & GERALD E. SOBELMAN (1999). Elliptic Curve Scalar Multiplier Design Using FPGAs. In *Cryptographic Hardware and Embedded Systems*, Ç. K. KOÇ & C. PAAR, editors, number 1717 in Lecture Notes in Computer Science, 257–268. Springer-Verlag. ISBN 3-540-66646-X. ISSN 0302-9743.

S. GAO, JOACHIM VON ZUR GATHEN & D. PANARIO (1995). Gauss periods, primitive normal bases, and fast exponentiation in finite fields. Technical Report 296-95, Dept. Computer Science, University of Toronto.

S. GAO & H. W. LENSTRA, JR. (1992). Optimal normal bases. *Designs, Codes, and Cryptography* **2**, 315–323.

SHUHONG GAO, JOACHIM VON ZUR GATHEN, DANIEL PANARIO & VICTOR SHOUP (2000). Algorithms for Exponentiation in Finite Fields. *Journal of Symbolic Computation* **29**(6), 879–889.

JOACHIM VON ZUR GATHEN & JÜRGEN GERHARD (1996). Arithmetic and Factorization of Polynomials over $\mathbb{F}_2$. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation ISSAC '96,* Zürich, Switzerland, Y. N. LAKSHMAN, editor, 1–9. ACM Press.

JOACHIM VON ZUR GATHEN & JÜRGEN GERHARD (2003). *Modern Computer Algebra*. Cambridge University Press, Cambridge, UK, 2nd edition. ISBN 0-521-82646-2. First edition 1999.

JOACHIM VON ZUR GATHEN & MICHAEL NÖCKER (2003). Computing special powers in finite fields. *Mathematics of Computation* **73**(247), 1499–1523. ISSN 0025-5718.

JOACHIM VON ZUR GATHEN & MICHAEL NÖCKER (2005). Polynomial and normal bases for finite fields. *Journal of Cryptology* **18**(4), 313–335.

JOACHIM VON ZUR GATHEN & JAMSHID SHOKROLLAHI (2005). Efficient FPGA-based Karatsuba multipliers for polynomials over $\mathbb{F}_2$. In *Selected Areas in Cryptography (SAC 2005),* BART PRENEEL & STAFFORD TAVARES, editors, number 3897 in Lecture Notes in Computer Science, 359–369. Springer-Verlag, Kingston, ON, Canada. ISBN 3-540-33108-5.

JOACHIM VON ZUR GATHEN & JAMSHID SHOKROLLAHI (2006). Fast arithmetic for polynomials over $\mathbb{F}_2$ in hardware. In *IEEE Information Theory Workshop (2006)*, 107–111. IEEE, Punta del Este, Uruguay.

J. GOODMAN & A. P. CHANDRAKASAN (2001). An energy-efficient reconfigurable public-key cryptography processor. *IEEE Journal of Solid-State Circuits* **36**(11), 1808–1820.

C. GRABBE, M. BEDNARA, J. SHOKROLLAHI, J. TEICH & J. VON ZUR GATHEN (2003a). FPGA Designs of parallel high performance $GF(2^{233})$ Multipliers. In *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS-03)*, volume II, 268–271. Bangkok, Thailand.

C. GRABBE, M. BEDNARA, J. SHOKROLLAHI, J. TEICH & J. VON ZUR GATHEN (2003b). A High Performance VLIW Processor for Finite Field Arithmetic. In *Proc. of The 10th Reconfigurable Architectures Workshop (RAW-03)*.

ROBERT GRANGER, DANIEL PAGE & MARTIJN STAM (2005). Hardware and Software Normal Basis Arithmetic for Pairing-Based Cryptogaphy in Characteristic Three. *IEEE Transactions on Computers* **54**(7), 852–860.

C. GREGORY, C. AHLQUIST, B. NELSON & M. RICE (1999). Optimal Finite Fields for FPGAs. In *Proceedings of the 9th International Workshop on Field Programmable Logic and Applications (FPL 99)*, number 1673 in Lecture Notes in Computer Science, 51–61. Springer-Verlag, Glasgow, UK.

J. H. GUO & C. L. WANG (1998). Hardware-efficient systolic architecture for inversion in $GF(2^m)$. *IEE Proc. -Comp. Digit. Tech.* **145**(4), 272–278.

VIPUL GUPTA, DOUGLAS STEBILA, STEPHEN FUNG, SHEUELING CHANG SHANTZ, NILS GURA & HANS EBERLE (2004). Speeding up secure Web Transactions Using Elliptic Curve Cryptography. In *The 11th Annual Network and Distributed System Security (NDSS) Symposium*. San Diego.

DARREL HANKERSON, JULIO LÓPEZ HERNANDEZ & ALFRED MENEZES (2000). Software Implementation of Elliptic Curve Cryptography Over Binary Fields. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, Ç. K. KOÇ & C. PAAR, editors, number 1965 in Lecture Notes in Computer Science, 1–24. Springer-Verlag. ISBN 3-540-41455-X. ISSN 0302-9743.

DARREL HANKERSON, ALFRED MENEZES & SCOTT VANSTONE (2003). *Guide to Elliptic Curve Cryptography*. Springer-Verlag. ISBN 0-387-95273-X.

JOHN L. HENNESY & DAVID A. PATTERSON (2003). *Computer Architecture A Quantitive Approach*. Morgan Kaufmann Publishers, 3rd edition. ISBN 1-55860-596-7.

M. JUNG, F. MADLENER, M. ERNST & S. HUSS (2002). A Reconfigurable Coprocessor for Finite Field Multiplication in $GF(2^n)$. In *Workshop on Cryptographic Hardware and Embedded Systems*. IEEE, Hamburg.

BURTON S. KALISKI & MOSES LISKOV (1999). Efficient Finite Field Basis Conversion Involving Dual Bases. In *Cryptographic Hardware and Embedded Systems*, Ç. K. KOÇ & C. PAAR, editors, number 1717 in Lecture Notes in Computer Science, 135–143. Springer-Verlag. ISBN 3-540-66646-X. ISSN 0302-9743.

A. KARATSUBA & YU. OFMAN (1963). Multiplication of multidigit numbers on automata. *Soviet Physics–Doklady* **7**(7), 595–596. Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July, 1962.

DONALD E. KNUTH (1998). *The Art of Computer Programming, vol. 2, Seminumerical Algorithms*. Addison-Wesley, Reading MA, 3rd edition. First edition 1969.

Ç. K. KOÇ & S. S. ERDEM (2002). Improved Karatsuba-Ofman Multiplication in $GF(2^m)$. US Patent Application.

Ç. K. KOÇ & B. SUNAR (1998). Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields. *IEEE Transactions on Computers* **47**(3), 353–356.

P: LEONG & I. LEUNG (2002). A microcoded elliptic curve processor using FPGA technology . *IEEE Transactions on VLSI* **10**(5), 550–559.

CHARLES VAN LOAN (1992). *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics (siam), Philadelphia. ISBN 0-89871-285-8.

JULIO LÓPEZ & RICARDO DAHAB (1999a). Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation. In *Cryptographic Hardware and Embedded Systems*, Ç. K. KOÇ & C. PAAR, editors, number 1717 in Lecture Notes in Computer Science, 316–327. Springer-Verlag. ISBN 3-540-66646-X. ISSN 0302-9743.

JULIO LÓPEZ & RICARDO DAHAB (1999b). Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$. In *Selected Areas in Cryptography*, STAFFORD TAVARES & HENK MEIJER, editors, number 1556 in Lecture Notes in Computer Science, 201–212. Springer-Verlag. ISBN 3-540-65894-7. ISSN 0302-9743.

JONATHAN LUTZ & ANWARUL HASAN (2004). High Performance FPGA based Elliptic Curve Cryptographic Co-Processor. In *International Conference on Information Technology: Coding and Computing (ITCC'04)*, volume 2, 486. IEEE.

UELI M. MAURER (1994). Towards the Equivalence of Breaking the Diffie-Hellman Protocol and Computing Discrete Logarithms. In *Advances in Cryptology: Proceedings of CRYPTO '94*, Santa Barbara CA, YVO G. DESMEDT, editor, number 839 in Lecture Notes in Computer Science, 271–281. Springer-Verlag. ISSN 0302-9743.

ROBERT J. MCELIECE (1987). *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers. ISBN 0-89836-191-6.

PETER L. MONTGOMERY (1987). Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation* **48**(177), 243–264.

PETER L. MONTGOMERY (2005). Five, Six, and seven-Term Karatsuba-Like Formulae. *IEEE Transactions on Computers* **54**(3), 362–369.

FRANÇOIS MORAIN & JORGE OLIVOS (1990). Speeding up the computations on an elliptic curve using addition-subtraction chains. *Informatique théorique et Applications/Theoretical Informatics and Applications* **24**(6), 531–544.

R. C. MULLIN, I. M. ONYSZCHUK, S. A. VANSTONE & R. M. WILSON (1989). Optimal normal bases in GF(p^n). *Discrete Applied Mathematics* **22**, 149–161.

MICHAEL NÖCKER (2001). *Data structures for parallel exponentiation in finite fields*. Doktorarbeit, Universität Paderborn, Germany.

JIMMY K. OMURA & JAMES L. MASSEY (1986). Computational method and apparatus for finite field arithmetic. *United States Patent 4,587,627* Date of Patent: May 6, 1986.

G. ORLANDO & C. PAAR (1999). A Super-Serial Galois Fields Multiplier for FPGAs and its Application to Public-Key Algorithms. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '99)*. Napa Valley, California, USA.

G. ORLANDO & C. PAAR (2000). A High-Performance Reconfigurable Elliptic Curve Coprocessor for $GF(2^m)$. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, Ç. K. KOÇ & C. PAAR, editors, number 1965 in Lecture Notes in Computer Science, 41–56. Springer-Verlag. ISBN 3-540-41455-X. ISSN 0302-9743.

MARTIN OTTO (2001). *Brauer addition-subtraction chains*. Diplomarbeit, University of Paderborn.

C. PAAR (1994). *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. Ph.D. thesis, Institute for Experimental Mathematics, University of Essen, Essen, Germany.

DONG JIN PARK, SANG GYOO SIM & PIL JOONG LEE (2003). Fast Scalar Multiplication Method Using Change-of-Basis Matrix to Prevent Power Analysis Attacks on Koblitz Curvesg. In *WISA 2003*, K. CHAE & M. YUNG, editors, number 2908, 474–488. Springer-Verlag, Berlin, Heidelberg.

PLANETMATH (2002). Lucas' theorem. Webpage. http://planetmath.org/encyclopedia/LucassTheorem.html.

A. REYHANI-MASOLEH & M. A. HASAN (2002). A new construction of Massey-Omura parallel multiplier over $GF(2^m)$. *IEEE Transactions on Computers* **51**(5), 511–520.

SHEUELING CHANG SHANTZ (2001). From Euclid's GCD to Montgomery Multiplication to the Great Divide. Technical Report TR-2001-95, Sun microsystems.

JOSEPH H. SILVERMAN (1986). *The Arithmetic of Elliptic Curves*. Number 106 in Graduate Texts in Mathematics. Springer-Verlag, New York.

N. P. SMART (2001). The Hessian form of an elliptic curve. In *Cryptographic Hardware and Embedded Systems CHES 2001*, Ç. K. KOÇ, D. NACCACHE & C. PAAR, editors, number 2162 in Lecture Notes in Computer Science, 118–125. Springer-Verlag. ISBN 3-540-42521-7. ISSN 0302-9743.

B. SUNAR & Ç. K. KOÇ (2001). An Efficient Optimal Normal Basis Type II Multiplier. *IEEE Transactions on Computers* **50**(1), 83–87.

NAOFUMI TAKAGI (1998). A VLSI Algorithm for Modular Division Based on the Binary GCD Algorithm. *IEICE Trans. Fundamentals* **E81-A**(5), 724–728.

ALFRED WASSERMANN (1993). Zur Arithmetik in endlichen Körpern. *Bayreuther Math. Schriften* **44**, 147–251.

A. WEIMERSKIRCH & C. PAAR (2003). Generalizations of the Karatsuba Algorithm for Efficient Implementations. Technical report, Ruhr-Universität-Bochum, Germany.

WIKIPEDIA (2006). Sierpinski triangle. Webpage.
http://en.wikipedia.org/wiki/Sierpinski_triangle.

HUAPENG WU (2000). On Computation of Polynomial Modular Reduction. Technical Report CORR2000-31, Centre for Applied Cryptographic Research (CACR), University of Waterloo, Canada.

XILINX (2005). *Virtex-II Platform FPGAs: Complete Data Sheet*. Xilinx. http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

List of symbols

$\mathbb{F}_{q^n}$	The finite field with q^n elements
$\mathsf{M}(n)$	Number of bit operations for multiplication of polynomials of length n
$\mathcal{Q}$	A fixed point on an elliptic curve
$-\mathcal{R}$	The additive inverse of the point $\mathcal{R}$ on an elliptic curve
$\mathcal{O}$	The point at infinity: the zero element of the group of points on an elliptic curve
$m\mathcal{R}$	The integer m times the point $\mathcal{R}$ on an elliptic curve
a, b	Parameters of an elliptic curve over a field of characteristic 2
ω	The root of the irreducible polynomial generating a polynomial basis
$f(x)$	The irreducible polynomial defining a polynomial basis
α	Normal element
w	The word-length of serial-parallel multipliers
$\mathcal{P}$	Polynomial basis
$H(x^k)$	The Hamming weight or the number of nonzero coefficients in the representation of x^k
$A_{LFSR}(n, \mathcal{P}, w)$	The area or the number of two-input gates in an LFSR multiplier for $\mathbb{F}_{2^n}$ of word-length w and in a polynomial basis $\mathcal{P}$
$D_{LFSR}(n, \mathcal{P}, w)$	The delay (or the minimum clock period) of an LFSR multiplier for $\mathbb{F}_{2^n}$ of word-length w and in a polynomial basis $\mathcal{P}$
T_X	The delay of an XOR gate
T_A	The delay of an AND gate
$\mathcal{K}_2$	The 2-segment Karatsuba method
$\mathsf{M}^{(2)}(n)$	The number of bit operations to multiply two n-bit polynomials by recursively applying the 2-segment Karatsuba method
$\mathcal{K}_3$	The 3-segment Karatsuba method
$\mathsf{M}^{(3)}(n)$	The number of bit operations to multiply two n-bit polynomials by recursively applying the 3-segment Karatsuba method
$\mathsf{M}^{(\mathcal{K}_2\mathcal{K}_3)}(6n)$	The number of bit operations to multiply two polynomials of length $6n$ by applying $\mathcal{K}_2$ on top of $\mathcal{K}_3$
$\mathsf{M}^{(\mathcal{K}_3\mathcal{K}_2)}(6n)$	The number of bit operations to multiply two polynomials of length $6n$ by applying $\mathcal{K}_3$ on top of $\mathcal{K}_2$
p	The characteristic of $\mathbb{F}_{q^n}$

$\mathbb{Z}_r^{\times}$	The group of units modulo r
$\#\mathbb{F}_{q^{nk}}$	The number of elements in $\mathbb{F}_{q^{nk}}$
β	A primitive rth root of unity in $\mathbb{F}_{q^{nk}}$
$(a_i^{(\mathcal{B})})_{1\leq i\leq n}$	The vector of the representation of the element a with respect to the basis $\mathcal{B}$
$a_i^{(\mathcal{B})}$	The ith entry in the vector of the representation of the element a with respect to the basis $\mathcal{B}$
$\mathbb{F}_q[x]^{\leq n}$	Polynomials in variable x over $\mathbb{F}_q$ with degree not larger than n.
φ_a	See Section 4.3. The polynomial representation of the element a
$(y_i)_{1\leq i\leq n}$	The vector of elements y_i for $1 \leq i \leq n$
$(A)_{i,j}$	The entry in the ith row and jth column of the matrix A
$L_{q,n}$	See Definition 6. The parameter q can be omitted.
ν_n	The linear mapping corresponding to $L_{q,n}$
$P_{q,n}$	See Definition 9. The parameter q can be omitted.
π_n	The linear mapping corresponding to $P_{q,n}$
L'_{p^r}	See Definition 6.
Θ_{p^r}	See Definition 11
$L_{p^r}^{(i,j)}$	The block in the ith row and jth column in the block representation of L_{p^r}
B_r	See Definition 13
$\mu_{add}(k)$	The number of $\mathbb{F}_q$-additions to multiply B_k by a vector in $\mathbb{F}_q^{p^k}$
$\mu_{mult}(k)$	The number of $\mathbb{F}_q$-multiplications to multiply B_k by a vector in $\mathbb{F}_q^{p^k}$
$\eta(r)$	The number of $\mathbb{F}_q$-additions to multiply L_{p^r} by a vector in $\mathbb{F}_q^{p^r}$, See Theorem 21
Φ_n	See Lemma 24

List of acronyms

AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Array
AT	Area-time
CLB	Configurable Logic Block
DES	Digital Encryption Standard
DFT	Discrete Fourier Transformation
DLP	Discrete Logarithm Problem
DSP	Digital Signal Processor
DSS	Digital Signature Standard
ECCo	Elliptic Curve Cryptography Co-Processor
ECDSA	Elliptic Curve Digital Signature Algorithm
FFT	Fast Fourier Transformation
FPGA	Field Programmable Gate Array
IDEA	International Data Encryption Algorithm
JNI	JAVA Native Interface
LFSR	Linear Feedback Shift Register
LUT	Look-Up Table
MO	Massey-Omura
PAR	Place and Route
PCI	Peripheral Component Interface
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RSA	(Ron) Rivest, (Fiat) Shamir, and (Leonard) Adleman
RTL	Register Transfer Level
SIM	Subscriber Identity Module
SHA	Secure Hash Algorithm
SoC	System on Chip
VHDL	VHSIC Hardware Description Language
VLSI	Very-Large-Scale Integration

Index