北京大学

博士研究生学位论文

题目：椭圆曲线加密体制的双有限域算法及其硬件实现

姓　　名：王　　健

学　　号：10548832

院　　系：信息科学技术学院微电子学系

专　　业：微电子学与固体电子学

研究方向：系统集成芯片设计及设计方法学

导师姓名：盛世敏　教授

二00八 年 六 月

摘　要

椭圆曲线加密算法（Elliptic Curve Cryptography，ECC）作为一种公钥体制加密算法，不但功能和 RSA 相同，而且和 RSA 相比具有加密强度高等诸多技术优点，因此应用越来越广泛。在这一背景下，对于椭圆曲线加密算法的研究也受到普遍重视。

论文重点研究椭圆曲线加密算法的改进，提出了一种支持椭圆曲线加密体制的双有限域算法。该算法可以完成素数域和二进制域上任意椭圆曲线的加密运算，并且模数 p 和取模多项式 f(x)可以任意选取。论文还提出了有限域运算单元的设计方法，这个运算单元可以同时完成素数域和二进制域上的所有运算，包括加法、减法、乘法、平方、求逆和除法。在对算法的改进和创新完成以后，论文讨论了对于改进后算法的硬件实现方案。点乘运算作为椭圆曲线加密算法的核心运算，是硬件实现设计的重点，针对改进后算法的特点论文创造性的提出了点乘运算电路的三级模块结构。把点乘电路分为点乘模块、点加倍点模块和有限域运算模块，并且针对三级模块的不同运算特点分别作了电路优化。

在整个的设计工作中，主要有 3 个创新点：

1、对于椭圆曲线加密运算芯片功能的扩展是本设计最重要的一点创新，我们所设计的点乘模块不仅可以同时完成素数域和二进制域两种有限域上的点乘运算，还可以完成这两种有限域上的加、减、乘、平方、求逆和除法运算，因此它可以应用到 ECC 和 RSA 等加密电路中；

2、椭圆曲线加密算法的核心运算点乘运算的算法创新，为了实现对两种有限域同时加密，我们首先对已有的算法进行了必要的改进，并且把各种算法进行合并，从而设计出了最后的点乘运算算法；

3、点乘运算电路结构上的创新，我们为了更好的实现点乘运算，把整个电路分为三级模块：点乘模块、点加倍点模块和有限域运算模块，并且针对不同模块的特点对每一级模块进行优化。

在完成了算法设计和电路设计之后，我们对最后的电路进行了全方位的仿真

与测试。对于点乘运算算法我们做了 VLSI 和 FPGA 两种硬件实现方案：VLSI 实现方案选择了中芯国际的 0.18 um 标准 CMOS 工艺，偏重于电路的功能仿真和对算法创新的验证；FPGA 实现方案则是在 Xilinx 公司生产的 Virtex2 系列的 XC2V3000 芯片上完成的，偏重于电路参数的测试和对电路结构创新的验证。论文第五章给出了 FPGA 的测试结果，当加密位数为 163 位，电路的时钟频率为 70MHz 时，完成素数域点乘运算只需要 4.23ms，完成二进制域点乘运算需要 2.29ms。本设计无论是在功能、速度还是面积上都具有很强的竞争力。仿真和测试结果的数据验证了本设计对于算法的创新以及电路结构的创造性改进都是成功的。

关键词：椭圆曲线加密算法，有限域，点乘，现场可编程门阵列实现

A Dual-Field Algorithm for Elliptic Curve Cryptosystem and its Hardware Implementation

WANG Jian (Microelectronics and Solid State electronics)

Directed by Professor SHENG ShiMin

Abstract

As Public-Key Cryptography, Elliptic Curve Cryptography (ECC) and RSA have the same function, but ECC has more technology advantages, so it has been used wider and wider. At the same time, the research of ECC has attracted a lot of attention.

In this paper, we improve the Algorithm of ECC and present a new Dual-Field ECC algorithm which supports both Galois fields GF(p) and GF(2^m) for arbitrary prime numbers and irreducible polynomials. An arithmetic unit is also designs, which can do the Galois field arithmetic operations of addition, subtraction, multiplication, squaring, inversion and division. After the design and innovate work of ECC algorithm we discuss the hardware implementation. As the kernel operation of ECC, the point multiplication is the focus point of the hardware implementation. The module of point multiplication is divided into 3 levels, the first level is point multiplication module, the second level is AD (addition and double) module, and the last level is Galois field arithmetic module. Then we can improve every module respectively.

In this paper, we present three innovations during the whole work:

1. We extend the function of the ECC circuit. Galois field arithmetic module is the most important part of the ECC circuit, which can support the point multiplication operation of both Galois fields GF(p) and GF(2^m) and all kinds of Galois field arithmetic operations, such as addition, subtraction, multiplication, squaring, inversion and division. The Galois field arithmetic module can not only been used for ECC, but also been used for RSA.

2. A new Dual-Field ECC Algorithm is proposed. The new Dual-Field ECC Algorithm can support Galois fields GF(p) and GF(2^m) and is suitable for hardware

implementation

3. We present a new construction of ECC point multiplication module originally, which is divided into 3 levels module.

After the work of algorithm design and circuit design, we emulate and test the ECC circuit. We achieve both VLSI implementation and FPGA implementation of the new Dual-Field ECC algorithm. The VLSI implementation using 0.18-μm CMOS standard cell library of SMIC puts the emphasis on the test of the algorithm improvement, and then the circuit has also been implemented on a Xilinx Virtex2 FPGA to test the innovation of ECC circuit construction. The FPGA implementation results of the ECC over 163-bit Galois fields are shown in this paper, which achieve a point multiplication time of 4.23ms at 70MHz over GF(p) and 2.29ms at 70MHz over GF(2^m). As the implementation and comparison results show, the proposed structure has strong advantages in function, speed and area. The emulation and test results show that the improvements and innovations of the algorithm and circuit structure are successful.

Keywords: ECC (Elliptic Curve Cryptography), Galois field, point multiplication, FPGA implementation

目　录

第一章　绪 论

椭圆曲线具有丰富多彩的历史，数学家对它的研究已经有一百多年了。人们利用椭圆曲线来解决各种不同的问题。一个例子是用来解决同余数问题，这个问题要求把表示某些直角三角形面积的正整数进行分类，而这些直角三角形的边长是有理数。另一个例子是用来证明 Fermat 大定理。Fermat 大定理叙述的是，当 n 是大于 2 的整数时，方程 $x^n + y^n = z^n$ 对于 x，y，z 无非零整数解。上面的两个例子都是椭圆曲线在数学上的应用。随着 Neal Koblitz 和 Victor Miller 在 1985 年分别独立地提出了利用椭圆曲线设计公钥密码体制的问题之后，在此后的 30 多年里椭圆曲线在密码学上的应用得到了充分的体现 [1]。自 1985 年以后，关于椭圆曲线密码安全性和有效实现的大批研究成果被发表出来 [2],[3]。1990 年以后，椭圆曲线密码开始得到商业界的认可，公认的标准化组织制定了椭圆曲线密码协议，一些公司在他们的安全产品中也采用了这些协议。

在下面的 1.1 节将介绍椭圆曲线加密算法研究的理论意义和现实意义；1.2 节将讲述椭圆曲线加密算法的发展以及国内外椭圆曲线加密算法的研究成果；1.3 节将讲述本文在研究椭圆曲线加密算法的设计及其实现中所运用的主要理论和方法。最后，在 1.4 节将给出论文的行文结构。

1.1 椭圆曲线加密算法研究的意义

随着椭圆曲线在公钥密码体制中的应用越来越广泛，人们对于椭圆曲线加密算法的研究也是越来越多，从算法到软件、硬件的实现都取得了不同程度的突破。在国内，椭圆曲线加密算法（Elliptic Curve Cryptography，ECC）的研究受到了普遍的重视，也出现了大量的研究成果，但是在椭圆曲线加密算法的实现方面，可应用的产品非常少。椭圆曲线加密算法和 RSA 加密算法的功能相同 [4]，但是拥有更多的技术优点，因此椭圆曲线加密算法在某些领域（如快速加密、密钥交换、身份验证、数字签名、移动通信的安全保密）的应用将取代 RSA 加密算法。

本文主要研究椭圆曲线加密算法的改进及其硬件实现方法，并将开发出一种以改进后的椭圆曲线加密算法为基础，在 FPGA（Field-Programmable Gate Array）

上实现的加密芯片，同时设计出有限域运算的 IP 核，这个 IP 核可以在 ECC 和 RSA 等加密算法的硬件电路中使用。目前在我国信息安全领域，公钥体制加密算法多采用 RSA 算法，采用 ECC 算法的相对较少，因此这个项目对我国加密技术的进一步发展具有巨大的推动作用。同时一款高效、兼容性强、可移植性好的椭圆曲线加密芯片也具有十分可观的市场应用前景。

1.2 椭圆曲线加密算法的发展历程

椭圆曲线加密算法是属于密码学的研究范畴，密码学是研究数据的加密及其变换的科学，它集数学、计算机科学、电子与通讯等诸多学科与一身，是研究编制密码和破译密码的技术科学。研究密码变化的客观规律，应用于编制密码以保守通信秘密的，称为编码学；应用于破译密码以获取通信情报的，称为破译学，总称密码学。

密码学是在编码与破译的斗争实践中逐步发展起来的，并随着先进科学技术的应用，已成为一门综合性的尖端技术科学。它与语言学、数学、电子学、声学、信息论、计算机科学等有着广泛而密切的联系。它的现实研究成果，特别是各国政府现用的密码编制及破译手段都具有高度的机密性。

在欧洲，公元前 405 年，斯巴达的将领来山得使用了原始的错乱密码；公元前一世纪，古罗马皇帝凯撒曾使用有序的单表代替密码；之后逐步发展为密本、多表代替及加乱等各种密码体制。二十世纪初，产生了最初的可以实用的机械式和电动式密码机，同时出现了商业密码机公司和市场。60 年代后，电子密码机得到较快的发展和广泛的应用，使密码的发展进入了一个新的阶段。

20 世纪 70 年代以来，一些学者提出了公开密钥体制，即运用单向函数的数学原理，以实现加、解密密钥的分离。加密密钥是公开的，解密密钥是保密的。这种新的密码体制，引起了密码学界的广泛注意和探讨。

自 19 世纪以来，由于电报特别是无线电报的广泛使用，为密码通信和第三者的截收都提供了极为有利的条件。通信保密和侦收破译形成了一条斗争十分激烈的隐蔽战线。1917 年，英国破译了德国外长齐默尔曼的电报，促成了美国对德宣战。1942 年，美国从破译日本海军密报中，获悉日军对中途岛地区的作战意图和兵力部署，从而能以劣势兵力击破日本海军的主力，扭转了太平洋地区的

战局。在保卫英伦三岛和其他许多著名的历史事件中，密码破译的成功都起到了极其重要的作用，这些事例也从反面说明了密码保密的重要地位和意义。

当今世界各主要国家的政府都十分重视密码工作，有的设立庞大机构，拨出巨额经费，集中数以万计的专家和科技人员，投入大量高速的电子计算机和其他先进设备进行工作。与此同时，各民间企业和学术界也对密码日益重视，不少数学家、计算机学家和其他有关学科的专家也投身于密码学的研究行列，更加速了密码学的发展。

椭圆曲线加密算法就是在这个密码学发展的大背景下应运而生的。它的出现标志着密码学的发展进入了崭新的篇章。

在前面已经提到过 20 世纪 70 年代以来，一些学者提出了公开密钥体制，事实上公钥密码体制的提出为椭圆曲线加密算法奠定了基础。公钥密码的概念是由 W. Diffie 和 M. Hellman 于 1976 年提出的。第一个实际的公钥密码是 1977 年 R. Rivest，A. Shamir 和 L. Adleman 提出的 RSA 公钥密码体制。RSA 密码是现在最著名的公钥密码之一，它的安全性建立在整数因子分解的困难性之上 [1]。

椭圆曲线密码（ECC）是 1985 年由 N. Koblitz 和 V. Miller 提出的。椭圆曲线密码属于公钥密码体制，它可以提供同 RSA 密码体制同样的功能。然而它的安全性建立在椭圆曲线离散对数问题（ECDLP）的困难性之上。现在求解 ECDLP 最好的算法具有全指数时间复杂度，与此不同，整数因子分解问题（RSA 密码的数学基础）却有亚指数时间算法。这意味着对于达到期望的安全程度，椭圆曲线密码可以使用较 RSA 密码更短的密钥。例如，普遍认为 163 位的椭圆曲线密码可以提供相当于 1024 位 RSA 密码的安全程度。由于密钥短而获得的优点包括加解密速度快、节省能源、节省带宽和节省存储空间 [1]。

正是由于椭圆曲线加密算法的诸多优点，人们看到了这种算法广阔的应用前景，因此近些年关于该算法的研究特别是应用上的学术成果层出不穷。在椭圆曲线加密算法的研究工作中，椭圆曲线密码编码学和密码破译学都取得了突破，本设计只讨论密码编码学。椭圆曲线密码编码学有如下几个热点：算法本身、实现方案和应用领域。其中就算法本身而言，研究的热点主要集中在椭圆曲线加密算法的核心运算点乘 [5], [6], [7]以及模逆运算 [8], [9], [10]这两方面，各种学术成果主要体现了平衡算法运算效率和算法所耗费的资源这个共同的思想。对于椭圆曲线加

密算法的改进，主要的目的就是使改进后的算法更加适用于软件或者硬件实现，因此对于算法实现方案的研究成果也非常多。一般情况下，二进制域上的椭圆曲线加密算法适合于硬件实现 [11], [12]，素数域上的椭圆曲线加密算法适合于软件实现 [13]。但是随着椭圆曲线加密算法的不断改进以及人们对于硬件实现的需求不断增加，也出现了大量的芯片实现了素数域上的椭圆曲线加密算法 [14], [15]。国内清华大学也有高质量的学术文章 [16]在国际上发表，另外要特别指出的是还有少量的研究工作 [17]是针对二进制域和素数域双有限域的硬件实现，复旦大学在这方面也有很好的研究成果 [18]。对于椭圆曲线加密算法的硬件实现来说，除了区分不同的有限域以外，对于硬件电路的实现方式上还有 VLSI 实现 [19], [20]和 FPGA 实现[21], [22]两种。这两种实现方式各有优缺点。最后一个研究热点就是椭圆曲线加密算法的应用，绝大部分研究成果都集中在利用其密钥短的优点，在需要低功耗和小存储空间的应用中采用 ECC 算法 [23]。将 ECC 密码应用到无线通信领域，国际国内都有这方面的研究成果 [24], [25]，还将 ECC 密码应用到智能卡 [26]以及其它产品当中 [27]。总之，ECC 密码凭借其密钥短的优点在很多便携式产品中被应用 [28], [29]。

通过上面的分析，可以预计 ECC 密码有着无比广阔的应用前景，其应用领域必将不断扩大。因此，可以同时实现二进制域和素数域的椭圆曲线加密芯片的研制将势在必行。针对目前国内外在这方面的研究成果较少的现状，本设计选择了这个突破口，并且重点研究并解决了以下两个重要问题：第一，如何高效的合并两个有限域上的加密算法；第二，如何减少芯片上的硬件资源，提高硬件资源的效率。

1.3 椭圆曲线加密算法的设计方法

椭圆曲线加密算法的设计主要包括两个方面 [30]：加密算法设计和软/硬件实现设计。对于算法设计来说，并不单纯是在数学层面上的计算，更重要的是与软/硬件实现相结合，设计出适合于软/硬件实现的算法。

椭圆曲线加密算法的加密和解密的过程是非常复杂的。图 1.1 给出了运算的层次，共分为四层：协议、点乘、椭圆曲线点加和倍点以及有限域算术 [1], [30]。在这四层运算中除了第一层协议有国际上通用的标准 [31], [32]，不需要进行设计以

外，其余三层所包含的运算都需要精心的设计，以保证椭圆曲线加密算法可以高效的进行加密和解密运算。在理论上，或者说在数学层面上点乘、椭圆曲线点加和倍点以及有限域算术都有一些学术界公认的最好算法。例如计算点乘的 AD 算法 [1]、计算点加和倍点的运算公式 [1]以及有限域乘法和除法的一些相关算法。但是这些算法并不一定适合于任何的应用情况，因此根据具体的应用条件设计者需要对已有的算法做出调整和改进。

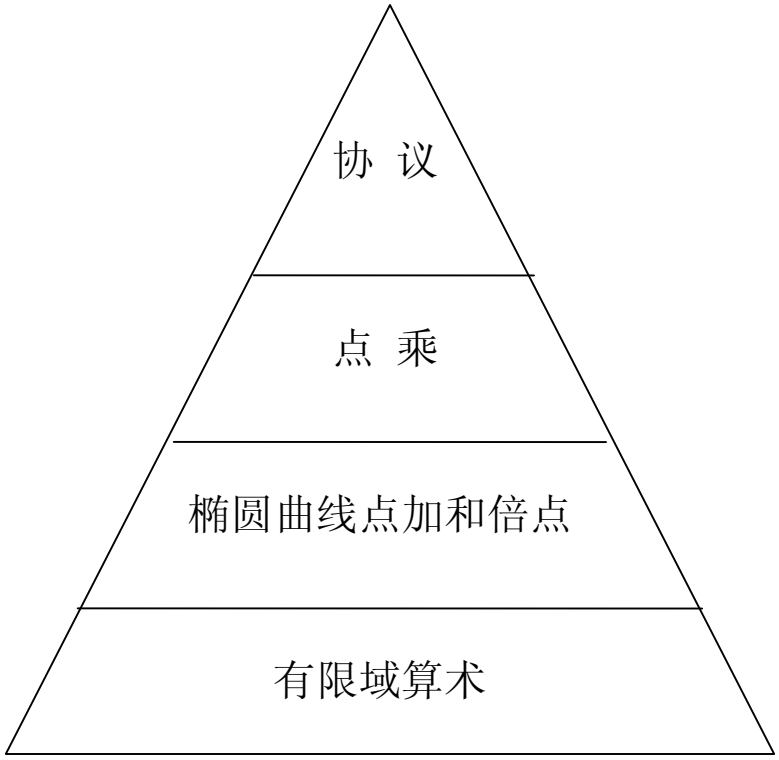

图 1.1　椭圆曲线加密算法的运算层次

设计的另外一个非常重要的方面就是软/硬件实现。如果是用软件来实现椭圆曲线加密算法，需要考虑的问题有：用什么语言来编写程序；该软件是在什么配置的硬件平台来运行。程序编写得好坏取决于它是否能够很好的还原算法本身，当然运行软件所基于的硬件平台也是非常重要的因素，二者共同决定椭圆曲线加密算法的运算效率。

通过硬件实现椭圆曲线加密算法比软件实现要复杂得多，成本也高。通常是在一些应用中，为了达到必需的安全级别而用软件实现的椭圆曲线加密方案无法

满足预期的性能要求时采用硬件实现。在这种情况下，设计和构造硬件加速器有利于满足性能的要求。通过硬件来实现椭圆曲线加密算法，主要有两种形式：VLSI（Very Large Scale Integration）实现和 FPGA（Field-Programmable Gate Array）实现。

随着 ECC 应用领域的不断扩大，越来越多的人开始研究软件和硬件的实现方案。总体来件，硬件的实现方案更加复杂，实现起来难度比较大。但是用硬件来实现 ECC 加密算法的趋势却不断增强，而且对于硬件的要求也是越来越高。通常认为当椭圆曲线算法的位数达到 163 位时，其安全性可以适合于绝大部分的应用领域 [1], [33]。

当前很多设计工作偏重于加密算法的速度，但是速度并不是硬件实现方案唯一考虑的问题，功耗以及成本都是需要考虑的重要因素。随着椭圆曲线加密算法及其实现的不断深入研究，以及椭圆曲线加密芯片在各个领域的广泛应用，人们对于椭圆曲线加密芯片的要求也不断提高。ECC 加密芯片的高速设计、低功耗设计等层出不穷。

在这种背景下，本文提出了一种椭圆曲线加密算法的核心运算——点乘的 FPGA 硬件实现方案，这个方案可以同时完成二进制域和素数域这两种有限域上的任意椭圆曲线的加密运算。在整个设计中，主要完成了以下工作：

1、通过查阅大量书籍和文献资料，对密码学、公钥密码体制、椭圆曲线加密体制以及有限域运算等领域有了深入、系统的了解；

2、掌握了椭圆曲线加密算法的核心运算——点乘的各种算法，并且通过对已有算法的改进和合并，设计出了高效的支持双域点乘运算的算法；

3、完成了点乘算法的硬件映射，成功地设计出点乘运算的电路模块；

4、完成了点乘电路模块的 FPGA 测试工作，得到了大量重要的电路参数，通过系统的分析和比较，证实我们的设计可以正确的完成椭圆曲线加密算法的加密和解密运算，其运算效率以及所耗费的硬件资源两方面都有很好的表现。

在整个地设计工作中，我们也取得了多个创新，主要的创新有：

1、对于椭圆曲线加密运算芯片功能的扩展是本设计最重要的一点创新。我们所设计的点乘模块不仅可以同时完成素数域和二进制域两种有限域上的点乘运算，还可以完成这两种有限域上的加、减、乘、平方、求逆和除法运算，因此

它可以应用的 ECC 和 RSA 等加密电路中；

2、我们对椭圆曲线加密算法的核心运算点乘运算做了重要的算法创新。为了实现对两种有限域同时加密，我们首先对已有的算法进行了必要的改进，并且把各种算法进行合并，从而设计出了最后的支持双域的点乘运算算法；

3、点乘运算电路的结构上也有重要的创新。为了更好的实现点乘运算，我们把整个电路分为三级模块：点乘模块、点加倍点模块和有限域运算模块，并且针对不同模块的特点对每一级模块进行优化。

关于我们的工作以及创新的具体内容将会在后续的章节中做出详细的讲述。

1.4 文章的行文结构

论文的第二章将会对整个密码学以及椭圆曲线加密算法的具体内容做出介绍；第三章讲述椭圆曲线加密算法所涉及的主要运算——有限域算术运算，这一章是本文的重点，本设计正是对有限域上的算术运算进行了改进从而提高了椭圆曲线加密算法的运算效率；在第四章将介绍椭圆曲线加密芯片中的核心运算模块——点乘模块的硬件设计；第五章介绍点乘模块的电路测试结果，将给出大量的电路参数以及对这些参数的分析和比较；最后，第六章对整个的研究工作做出总结与展望。

第二章　椭圆曲线加密算法

密码学是研究数据的加密及其变换的科学，它集数学、计算机科学、电子与通讯等诸多学科与一身。长期以来，由于密码技术的隐秘性，应用一般局限于政治、经济、军事、外交、情报等重要部门，密码学鲜为人知。进入 20 世纪 80 年代后，随着计算机网络，特别是因特网的普及，密码学得到了广泛的重视，如今密码技术不仅服务于信息的加密和解密，还是身份认证、访问控制、数学签名等多种安全机制的基础。

椭圆曲线的相关研究开始于 19 世纪，至今已有 150 多年了，如著名的 Weierstrass 方程。Kail Weierstrass 就是 19 世纪的一位数学家，其在数论领域做出了一些开拓性的工作。椭圆曲线在费马大定理的证明中起到了很重要的作用 [1]；其应用于密码学领域，是始于 1985 年 N. Koblitz [3]和 Miller [2]的工作。

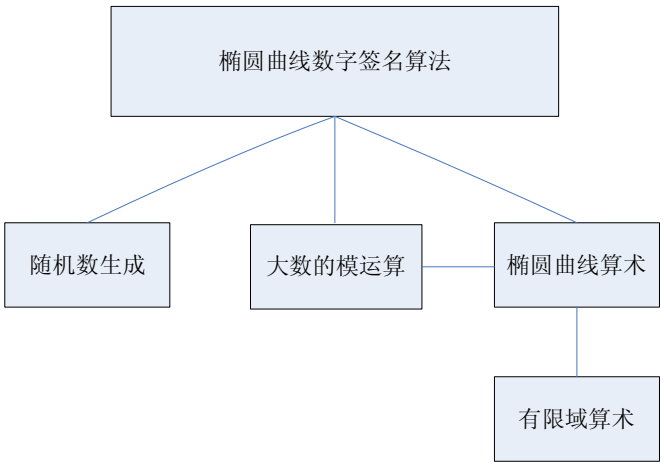

图 2.1　ECDSA 模型框图

图 2.1 给出了椭圆曲线数字签名算法（ECDSA）的协议模型框架 [1]。从这个图可以看出椭圆曲线的算术运算不仅仅依赖于域的算术运算，而且还依赖于大数的模运算。ECDSA 还应用杂凑函数和一些模运算，但是最费时的运算步骤还是椭圆曲线的算术运算。椭圆曲线算术运算是椭圆曲线加密算法加/解密的核心

运算，而随机数生成和杂凑函数只是应用于算法的密钥生成，并不是本设计讨论的范围。

2.1 节介绍密码学的发展背景。2.2 节介绍密码体制的分类。2.3 节为椭圆曲线概述。2.4 节介绍椭圆曲线上的加法运算。2.5 节介绍椭圆曲线上最为重要的运算——点乘。如何把椭圆曲线应用到公钥密码体制中，2.6 节对于椭圆曲线加密体制的介绍将解答这个问题。在 2.7 节将讲述椭圆曲线加密算法的应用。最后 2.8 节是本章小结。

2.1 密码学发展背景

密码学的发展大致可分为 4 个阶段 [34]：第一个阶段是指第一次世界大战之前的古典密码术（手工操作密码），它有几千年的历史；第二个阶段是指第一次世界大战爆发至第二次世界大战结束期间的机器密码时代；第三个阶段是以 C. E. Shannon 在 1949 年发表的文章“The Communication Theory of Secrecy System”（《保密系统的通信理论》）为起点的传统密码学；第四个阶段是以 W. Diffie 和 M. E. Hellman 在 1976 年发表的文章“New Directions in Cryptography”（《密码学新方向》）为起点的现代公钥密码学 [35]。

2.1.1 古典密码术

古典密码的特征主要是以纸和笔进行加密与解密操作的密码术时代，这时密码还远没有成为一门科学，仅仅是一门技艺或者技术。古典密码的基本技巧都是较简单的代替、置换、或二者混合使用。古典密码的历史最早可以追溯到四千多年前雕刻在古埃及法老纪念碑上的奇特象形文字。不过这些奇特的象形文字记录不可能是用于严格意义上的保护秘密信息的，而更可能仅是为了神秘、娱乐等目的。

2.1.2 近代机器密码

在手工密码时代，人们通过纸和笔对字符进行加密和解密，速度不仅慢，而且枯燥乏味、工作量繁重。因此手工密码算法的设计受到一定的限制，不能设计

很复杂的密码。随着第一次世界大战的爆发，工业革命的兴起，密码术也进入了机器时代。与手工操作相比，机器密码使用了更加复杂的加密手段，同时加密解密效率也得到了提高。在这个时期虽然加密设备有了很大的进步，但是还没有形成密码学理论。加密的主要原理仍然是代替、置换、或者是二者的结合。

2.1.3 传统密码学

1949 年以前的密码还不能称为是一门学科，而只能称为是一种密码技术或密码术。

1948 年，Shannon（C. E. Shannon）在 Bell 系统技术期刊（“the Bell System Technical Journal ”）上发表了他的论文“ A Mathematical Theory of Communication”，该论文应用概率论的思想来阐述如何最好地加密要发送的信息。1949 年，Shannon 又在 Bell 系统技术期刊上发表了他的另一篇著名论文“The Communication Theory of Secrecy Systems”，这篇文章标志着传统密码学（理论）的真正开始。在该文中，Shannon 首次将信息论引入了密码研究中。他利用概率统计的观点，同时引入熵（entropy）的概念，对信息源、密钥源、接收和截获的密文，以及密码系统的安全性进行了数学描述和定量分析，并提出了通用的秘密密钥密码体制模型（即对称密码体制的模型），从而使密码研究真正成为了一门学科。Shannon 的这两篇文章，加之在信息与通信理论方面的一些其他工作，为现代密码学及密码分析学奠定了坚实的理论基础。

此后直到 20 世纪 70 年代中期，密码学的研究基本上是在军事和政府部门秘密的进行，所取得的研究进展不大，几乎没有见到什么研究成果公开发表。

1974 年，IBM 公司应美国国家标准局 NBS（现已更名为国家标准与技术研究局 NIST）的要求，提交了基于 Lucifer 密码算法的一种改进算法，即数据加密标准 DES（data encryption standard）算法。1976 年底 NBS 正式颁布 DES 作为联邦信息处理标准 FIPS（Federal Information Processing Standard），此后 DES 在政府部门以及民间商业部门等领域得到了广泛的应用 [36], [37], [38]。自此，传统密码学的理论与应用研究真正步入了蓬勃发展的道路。

自 Caesar 密码至 NBS 颁布的 DES，所有这些密码系统在加密与解密时所使用的密钥或电报密码本均相同，通信各方在进行秘密通信前，必须通过安全渠道获得

同一密钥。这种密码体制称为传统密码体制或对称密码体制。自 DES 颁布后出现的主要传统密码体制有：三重 DES、IDEA、Blowfish、RC5、CAST-128，以及 AES 等 [34]。

2.1.4 现代公钥密码学

1976 年以前的所有密码系统均属于对称密码体制。但是在 1976 年，W. Diffie 与 M. E. Hellman 在期刊“IEEE Transactions on Information Theory”上发表了一篇著名的论文“New Directions in Crpytography” [35]，为现代密码学的发展打开了一个崭新的思路，开创了现代公钥密码学。

在这篇文章中，W. Diffie 与 M. E. Hellman 首次提出设想，在一个密码体制中，不仅加密算法本身可以公开，甚至用于加密的密钥也是可以公开的。也就是说，一个密码体制可以有两个不同的密钥：一个是必须保密的密钥，另一个是可以公开的密钥。若这样的公钥体制存在，就可以将公开密钥像电话簿一样公开，当一个用户需要向其他用户传送一条秘密信息时，就可以先从公开渠道查到该用户的公开密钥，用此密钥加密信息后将密文发送给该用户。此后该用户用他保密的密钥解密接收到的密文，即得到明文，而任何第三者则不能获得明文 [35]。这也就是公钥密码体制的构想。虽然他们当时没有提出一个完整的公钥密码体制，但是他们认为这样的密码体制一定存在。同时在该文中，他们给出了一个利用公开信道交换密钥的方案，即后来以他们的名字命名的 Diffie-Hellman 密钥交换协议 [35]。利用此协议，通信各方不需要通过一个安全渠道进行密钥传送就可以进行保密通信。

就在 W. Diffie 与 M. E. Hellman 提出他们的公钥密码思想大约一年后，麻省理工学院的 Ron Rivest、Adi Shamir 和 Len Adleman 向世人公布了第一个这样的公钥密码算法，即以他们的名字的首字母命名的具有实际应用意义的 RSA 公钥密码体制 [39]，并且很快被接受成为 ISO/IEC（International Standards Organizational’s International Electrotechnical Commission）、ITU-T（International Telecommunications Union’s Telecommunication Standardization Sector）、ANSI（American National Standards Institute），以及 SWIFT（Society for Worldwide Interbank Financial Telecommunications）等国际化标准组织采用的公钥密码标准

算法，并得到了广泛应用。此后不久，人们又相继提出了 ElGamal [40]和椭圆曲线密码体制（ECC）等公钥密码体制 [2], [3]。随着电子计算机等科学技术的进步，公钥密码体制的研究与应用得到了快速的发展。

2.2 密码体制分类

密码学就其发展而言，主要经历了前面介绍的 4 个阶段：古典密码术阶段、机器密码阶段、传统密码学阶段、和现代公钥密码学阶段。前两个阶段都是基于字符替换的密码，现在已经很少使用，但它代表了密码的起源，虽然那时密码学并没有成为一门真正的学科，当然也没有完备的密码体制。

从传统密码学阶段开始一直到现在，密码学一直以来是作为一门学科来研究，并建立了非常完备的密码体制。按不同的划分标准或者方式，密码体制可以分为多种形式。如果按照密钥的管理方式可以把密码体制分为对称密钥密码体制、公开密钥密码体制以及近几年提出来的量子密码体制三大类 [34], [41]。

2.2.1 对称密钥密码体制

对称密钥密码体制是从传统的简单替换发展而来的。传统的密码体制所用的加密密钥和解密密钥相同，或实质上等同（即从一个可以推出另一个），我们称其为对称密钥、私钥或单钥密码体制。对称密钥密码体制不仅可以用于数据加密，也可以用于消息的认证。

按加密模式来分，对称算法又可分为序列密码和分组密码两大类。序列密码每次加密一位或一字节的明文，也称为流密码。序列密码是手工和机械密码时代的主流方式。分组密码将明文分成固定长度的组，用同一密钥和算法对每一块加密，输出也是固定长度的密文。最有影响的对称密钥加密算法是 1977 年美国国家标准局颁布的 DES 算法 [38]。其次还有 CAST，RC5，3EDS，SAFER，Blowfish，FEAL，IDEA，Skipjack 等，它们都属于分组密码 [34], [41]。

对称密钥密码体制的优点是安全性高且加、解密速度快，其缺点是密钥的分发与管理难度大。进行保密通信之前，双方必须通过安全信道传送所有的密钥，这对于相距较远的用户可能要付出较大的代价，甚至难以实现。因为对于对称密钥密码体制，通信的双方必须就密钥的秘密性和真实性达成一致，即要求信道即

保密且保真。与之相比，公钥密码体制只要求信道是保真的，因此可以很好的解决了密钥的分发与管理问题（如图 2.2）[1]。

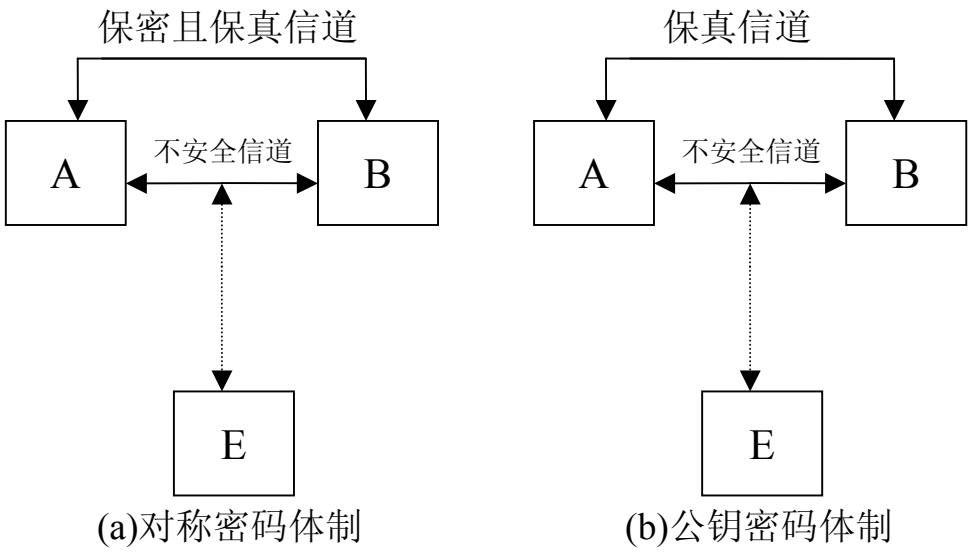

图 2.2　对称与公钥密码体制

2.2.2 公开密钥密码体制

若加密密钥和解密密钥不相同，从其中一个难以推出另一个，则称为双钥密码体制或公开密钥密码体制，这是 1976 年由 Diffie 和 Hellman 等学者所开创的新密码体制 [35]。

对于单钥体制存在的问题，采用双钥体制则可以完全克服，特别是多用户通信网，双钥体制可以明显减少多用户之间所需的密钥量，从而便于密钥管理。采用双钥密码体制的主要特点是将加密和解密分开，因而可以实现多个用户加密的消息只能由一个用户解读，或只能由一个用户加密消息而使多个用户可以解读[34], [41]。

在使用双钥体制时，每个用户都有一对预先选定的密钥对：一个是可以公开的，叫做公钥，以 k_1 表示，另一个则是秘密的，叫做私钥，以 k_2 表示。公开的密钥 k_1 可以像电话号码一样进行注册公布，因此我们把这种密码体制通常称为

公钥体制（public key system），其通信模型如图 2.3 [1]。

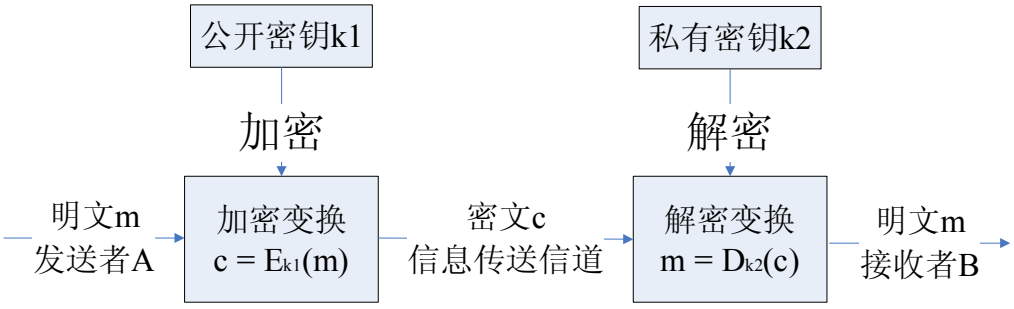

图 2.3　双钥密码体制的通信模型

在公钥密码体制中，密钥对的选择要保证从公钥求出私钥等价于要求解一个困难的计算问题。构成常用公钥密码基础的困难问题有如下的数论问题 [1]：

1、整数的因子分解问题，其困难性是 RSA 公钥密码安全性的基础 [39]。

2、 离散对数问题，其困难性是 ElGamal 公钥密码及其变体（如数字签名算法）安全性的基础 [40]。

3、 椭圆曲线离散对数问题，其困难性是椭圆曲线公钥密码体制安全性的基础 [2], [3]。

2.2.3 量子密码体制

1970 年，美国科学家威斯纳首次提出了量子密码技术，威斯纳当时的想法是利用单量子态制造不可伪造的“电子钞票”。但这个设想的实现需要长时间保存单量子态，不太现实，因此很长时间以来，量子加密技术不受重视。随后，贝内特和布拉萨德两位学者在研究中发现：单量子态虽然不好保存但可以用于传输信息。1984 年，他们提出了第一个量子密钥分配方案 BB84 协议，标志着量子密码体制研究真正的开始。

量子密码是以量子力学和密码学为基础，利用量子物理学原理实现密码体制的一种新型密码体制，与当前大多使用的经典密码体制不一样的是，量子密码利用信息载体的物理属性实现。目前，量子密码中用于承载信息的载体包括光子、压缩态光信号、相干态光信号等。这些信息载体可通过多个不同的物理量来描述，

比如偏振、相位、振幅等。当前量子密码实验中，大多采用光子作为信息的载体。利用光子的偏振属性进行编码，由于在长距离的光纤传输中，光子的偏振性因退化而受到影响，因此也有利用光子的相位进行编码的方法。

量子密码体制的理论基础是量子物理定理，而物理定理是物理学家经过多年的研究与论证得出的结论，有可靠的理论依据，且不论在何时都是不会变的，因此理论上，依赖于这些物理定理的量子密码也是不可攻破的，量子密码体制是一种绝对保密的密码体制。

目前量子密码体制正处于试验阶段，并且其密钥分配严重依赖于光学材料，距离实际应用仍有一定距离 [42]。

2.3 椭圆曲线概述

椭圆曲线不是通常所指的椭圆，所谓椭圆曲线是指亏格为 1 的平面代数曲线。一般地，可以用 Weierstrass 方程描述 [1],[43], [44], [45], [46]：

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

其中 $a_1, a_2, a_3, a_4, a_6 \in F$ 且$\triangle \neq 0$，$\triangle$是方程的判别式。F 是一个域，可以是有理数域、复数域，还可以是有限域 $GF(p^r)$。满足上面方程的点(x, y)就构成椭圆曲线。

椭圆曲线有很多优美的结果，而密码学界使用椭圆曲线的目的在于：椭圆曲线上可以提供无数的有限 Abel 群，并且由于这种群的结构丰富、易于实际计算，从而可以用于构造密码算法。

Weierstrass 方程的表达形式是复杂的，在密码学中，我们常用该方程的简化形式：

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

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

上面这两种形式在密码学中最为常用，它们的判别式可以表示为：$\triangle = 4a^3 + 27b^2 \neq 0$ 和$\triangle = b \neq 0$。$y^2 = x^3 + ax + b$ 这个式子通常作为素数域 F_p 上的椭圆曲线方程，$y^2 + xy = x^3 + ax^2 + b$ 通常为二进制域 F_{2^m} 上的椭圆曲线方程。素数域可以用 F_p

或者 GF(p)来表示，二进制域可以用 F_{2^m} 或者 GF(2^m)来表示，这两个概念将在后面的章节做出详细介绍。

密码学家通常把椭圆曲线划分为 F_p、F_2 多项式形式，以及 F_2 最优法线等形式。当然密码学中使用的是曲线的离散点，并且只用整数点，而不是用非整数点。所有的这些整数点都包含在一个区域内部，区域越大，曲线的安全性越高；区域越小，安全性越低，但计算速度越快。

2.4 椭圆曲线上的加法

密码学中探讨椭圆曲线，都是基于某个有限域 [47]讨论，例如基于 Z_p，p 为素数。只有在有限域基础上，才有可能在信息变换中进行精确计算。为了提高计算速度，通常是基于有限域 F_p（p 为素数）或者 F_{2^m}（m 为正整数）讨论椭圆曲线 [48], [49]。这两种有限域是椭圆曲线上最常见的，通常称 F_p 为素数域，F_{2^m} 为二进制域。

下面介绍一下椭圆曲线上的“加法”。如图 2.4 所示，在椭圆曲线上取任意两点 P、Q，通常过 P、Q 两点的直线若与椭圆曲线有第三个交点，则把该点记为-R，点-R 关于 x 轴的对称点 R 也在该椭圆曲线上。通常就定义“加法”：

$$P + Q = R$$

我们定义椭圆曲线“加法”，是为了用这些点构造出一个加法群，以用于密码算法。其中一个显然的问题是：该加法群的单位元是什么？从图 2.4 可以看出，如果直线通过点 R 及其对称点-R（即直线与 y 轴平行），显然直线与该椭圆曲线没有第三个交点，那么它们的和：R + (-R)等于什么呢？规定 R + (-R) = O，O 称为零点，表示此时直线在无穷远处与曲线有交点，O 也就等价于加法群中的单位元——零点。

这样，在实数域下，椭圆曲线上的所有点，再加上一个无穷远点——零点，构成一个加法群。当然，我们关心的是曲线上整数点构成的加法群。

在定义完零点以后我们开始讨论基于有限域 F_p（p 为素数）或者 F_{2^m}（m 为正整数）椭圆曲线上加法的具体公式。下面约定几个符号。E(F_p)和 E(F_{2^m})分别

表示基于 F_p 和 F_{2^m} 上的椭圆曲线；O 表示零点——无穷远点；点 P 对应一对坐标(x，y)，其对称点用-P 表示，-P 的坐标为(x，-y) [1], [44]。

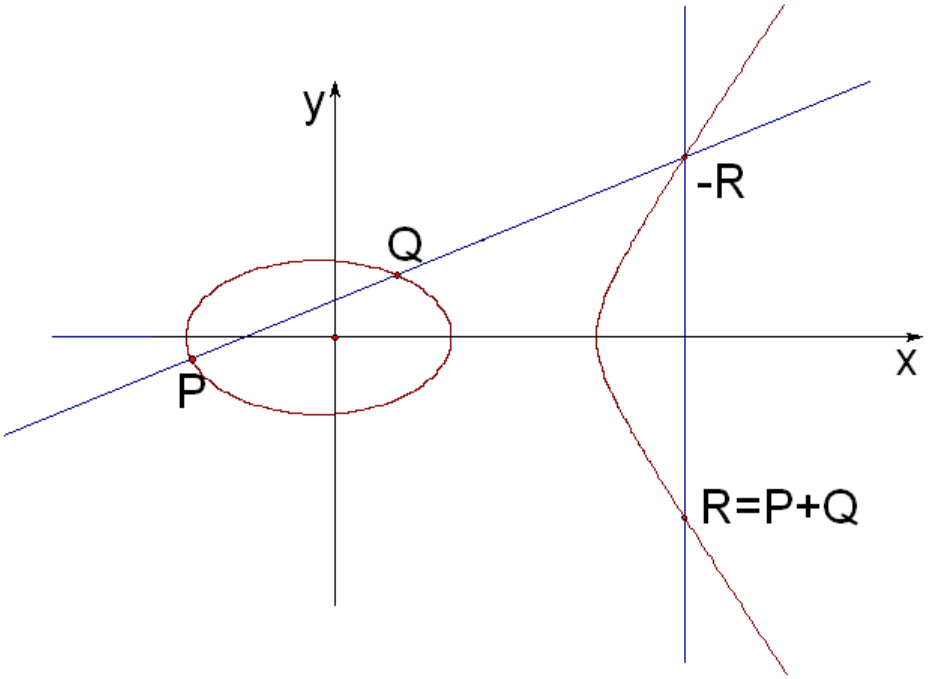

图 2.4　椭圆曲线“加法”示意图

2.4.1 素数域 F_p 上的椭圆曲线加法

先讨论基于素数域 F_p（p 为素数）上的椭圆曲线加法，定义基于 Z_p 的椭圆曲线为：

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

并且有：a、b $\in Z_p$，$\triangle = 4a^3 + 27b^2 \neq 0 \bmod p$。

所谓的加法，其加数与被加数都属于一个点集合，该点集合是由椭圆曲线上的点与零点构成的。下面分情况给出加法公式 [1], [44], [48], [49]：

1、与零点 O 相加。对于所有的点 P $\in E(F_p)$，有 P + O = O + P = P。

2、与对称点相加。对于所有的点 P $\in E(F_p)$，有 P + (-P) = (-P) + P = O。

可以说明如下：P = (x, y)，其对称点-P = (x, -y)，所以过该两点的直线与 y 轴平行，与曲线相交于无穷远点。P 和-P 互为逆元，在加法群中互称为相反数。因为椭圆曲线关于 x 轴对称，所以椭圆曲线群中的每一个元素都有逆元。

3、两相异点相加（点加）。对于任意两点 P,Q， $P,Q \in E(F_p)$，且 $P \neq \pm Q$。令 $P=(x_1,y_1)$， $Q=(x_2,y_2)$， $P+Q=(x_3,y_3)$，那么有：

$$x_3=(\frac{y_2-y_1}{x_2-x_1})^2-x_1-x_2$$

$$y_3=(\frac{y_2-y_1}{x_2-x_1})(x_1-x_3)-y_1$$

4、两相同点相加（倍点）。对于任意点 P，$P=(x_1,y_1)\in E(F_{2^m})$，要求 $P\neq -P$，那么 $P+P=2P=(x_3,y_3)$，那么有：

$$x_3=(\frac{3x_1^{\ 2}+a}{2y_1})^2-2x_1$$

$$y_3=(\frac{3x_1^{\ 2}+a}{2y_1})(x_1-x_3)-y_1$$

注意公式里面的参数 a 与椭圆曲线 $y^2 = x^3 + ax + b$ 里的参数 a 为同一个数值。在图像上表示，这个公式的含义就是过 P 点的曲线的切线，与曲线的交点为-2P。

$P\neq -P$ 表示曲线 E 上过点 P 的切线不与 y 轴平行。考虑相反情况，曲线 E 上某点 P，如果过该点的切线与 y 轴平行，结合椭圆曲线图像即有 P = -P，可以认为该切线与椭圆曲线还相交于无穷远点，即 P + (-P) = O。

2.4.2 二进制域 F_{2^m} 上的椭圆曲线加法

一般情况下，定义基于 F_{2^m} 上的椭圆曲线方程为：

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

其中 a、b∈ F_{2^m}，判别式△ = b ≠ 0。那么基于 F_{2^m} 上的椭圆曲线 E(F_{2^m})的点(x, y)满足：x、y∈ F_{2^m}，当然 O 定义为无穷远点（零点）。

在此简要给出 F_{2^m} 上的椭圆曲线的加法公式 [1], [44], [48], [49]：

1、与零点 O 相加。对于所有的点 P ∈ E(F_{2^m})，有 P + O = O + P = P。

2、与对称点相加。对于所有的点 P ∈ E(F_{2^m})，有 P + (-P) = (-P) + P = O。可以说明如下：P = (x, y)，其对称点-P = (x, x + y)。

3、两相异点相加（点加）。对于任意两点 P,Q， $P,Q \in E(F_{2^m})$，且 $P \neq \pm Q$。令 $P=(x_1,y_1)$， $Q=(x_2,y_2)$， $P+Q=(x_3,y_3)$，那么有::

$$x_3=(\frac{y_1+y_2}{x_1+x_2})^2+\frac{y_1+y_2}{x_1+x_2}+x_1+x_2+a$$

$$y_3=(\frac{y_1+y_2}{x_1+x_2})^2(x_1+x_3)+x_3+y_1$$

4、两相同点相加（倍点）。对于任意点 P，$P=(x_1,y_1)\in E(F_{2^m})$，要求 $P\neq -P$，那么 $P+P=2P=(x_3,y_3)$，那么有：

$$x_3={x_1}^2+\frac{b}{{x_1}^2}$$

$$y_3={x_1}^2+(x_1+\frac{y_1}{x_1})x_3+x_3$$

2.5 椭圆曲线上的点乘

这部分我们讨论椭圆曲线上的点乘 kP 的计算方法，其中 k 是一个整数，P 是定义在有限域 F_p 或 F_{2^m} 上的椭圆曲线 E 上的一个点，点乘的运算速度决定着椭圆曲线密码体制的运算速度。需要强调的是在这里所讨论的技术不依赖于任何具体的曲线结构，它是适用于任何椭圆曲线的，因此具有普遍意义。在此先给出点乘 kP 的计算公式：

$$\underbrace{P+P+\cdots+P}_{k}=kP$$

在此给出椭圆曲线上阶的定义：设点 P 是椭圆曲线 E 上的一点，如果存在最小的整数 n，使得 nP = O，其中 O 表示无穷远点（零点），那么称点 P 的阶为 n。曲线上的某些点不一定存在有限的阶，但是在密码学中主要关心的是曲线 E 上有限阶的点。根据上面对阶的定义可知，点乘 kP 中的整数 k 要满足以下条件：$1\le k\le n-1$。

假设在椭圆曲线 $E(F_p)$上有两点 P 和 Q，它们的阶为 n，乘数 k 随机地从 $1 \le k \le n-1$ 内选择。k 的二进制表示为 $(k_{m-1},\cdots,k_2,k_1,k_0)_2$。下面介绍两种最为常用的计算点乘的方法。

2.5.1 二进制方法

二进制方法是最基本也是最常用的计算点乘的算法，这种方法可以从二进制数串的最高位和最低位两个方向进行计算。

算法 1 计算点乘的从右向左的二进制方法

输入：$k=(k_{m-1},\cdots,k_2,k_1,k_0)_2, P \in E(F_p)$。

输出：kP。

1、Q = 0。

2、对于 i 从 0 到 m – 1，重复执行

2.1 若 $k_i = 1$，则 Q = Q + P。

2.2 P = 2P。

3、返回（Q）。

算法 2 计算点乘的从左向右的二进制方法

输入：$k=(k_{m-1},\cdots,k_2,k_1,k_0)_2, P \in E(F_p)$。

输出：kP。

1、Q = 0。

2、对于 i 从 m – 1 到 0，重复执行

2.1 Q = 2Q。

2.2 若 $k_i = 1$，则 Q = Q + P。

3、返回（Q）。

算法 1 和算法 2 是在很多文献和书籍中比较常见的算法 [50], [51]，我们对这个

算法做了重要的改进。

算法 3 改进后的点乘二进制方法

输入：$k=(k_{m-1},\cdots,k_2,k_1,k_0)_2$，$P$ 为有限域（素域或二进制域）上的任意一点。

输出：kP。

1、$Q=0$。

2、重复执行

2.1 若 $k_0=1$，则 $Q=Q+P$。

2.2 $P=2P$。

2.3 $k=k>>1$。

2.4 若 $k=0$，则执行步骤 3。

3、返回（Q）。

算法 3 的改进主要体现在步骤 2.3 和 2.4，通过对 k 的右移操作和对 k 是否为“0”的判断，来控制点加和倍点的运算次数。这种控制比固定的执行 m 次操作要灵活得多，并且增加的移位和判断运算对于点乘运算速度的影响非常小。

下面我们简要分析一下二进制算法的运算复杂度。k 的二进制表示中“1”的个数的期望值为 m/2（设 k 的二进制表示位 m 位），因而此算法的期望运行时间近似为 m/2 次点加运算和 m 次倍点运算，若用 A 表示点加运算、D 表示倍点运算，则期望运行时间可以表示为：

$$\frac{m}{2}A+mD$$

令 M 表示有限域乘法（有限域的运算将在下一章详细介绍），S 表示有限域的平方，I 表示有限域的求逆运算。通过对素数域 F_p 上加法公式的分析可知，点加运算需要做一次平方、两次乘法和一次求逆运算，倍点运算需要做两次乘方、两次乘法和一次求逆运算，因此在素数域 F_p 上用域运算表示的运行时间为：

$$2.5mS+3mM+1.5mI$$

用同样的方法，在二进制域 F_{2^m} 上运算时间可以表示为：

$$1.5mS + 3mM + 1.5mI$$

2.5.2 二进制 NAF 方法

在介绍这种方法之前先介绍一些关于非相邻表示型（NAF）的相关概念 [1]。在二进制域中，设 P = (x, y)∈E(F_{2^m})，则-P = (x, x + y)；而在素域中，设 P = (x, y)∈E(F_p)，则-P = (x, -y)。因此椭圆曲线点的减法与点的加法一样有效。这促使我们使用 k 的带符号数字表示 k，即

$$k = \sum_{i=0}^{m-1} k_i 2^i \qquad k_i \in \{0, \pm 1\}$$

一种特别有用的带符号数字表示就是非相邻表示型（NAF）。其定义如下：

一个正整数 k 的非相邻型是表达式

$$k = \sum_{i=0}^{m-1} k_i 2^i \qquad k_i \in \{0, \pm 1\}$$

且 $k_{m-1} \neq 1$，并且没有两个连续数字 k_i 是非零的。NAF 的长度是 l。

当 k 为正整数时，NAF 的性质有：

（1） k 有唯一的 NAF，并记为 NAF(k)。

（2） NAF(k)在 k 的所有带符号表示中具有最少的非零位。

（3） NAF(k)的长度最多比 k 的二进制表示的长度大 1。

（4） 若 NAF(k)的长度是 m，则 $2^m/3 < k < 2^{m+1}/3$。

（5） 所有长度为 m 的 NAF 中非零数字的平均个数约为 m/3。

利用算法 4 [1]可以有效地计算 NAF(k)。NAF(k)的各位数字可以通过连续用 2 去除 k 且允许余数取 0 或±1 来得到。若 k 是奇数，则余数 r∈{-1, 1}，以使商(k – r) / 2 是偶数，这将确保下一个 NAF 数字是 0。

算法 4 计算一个正整数的 NAF

输入：一个正整数 k。

输出：NAF(k)。

1、i = 0。

2、当 $k \geq 1$ 时，重复执行

2.1 若 k 为奇数，则 $k_i = 2 - (k \bmod 4)$, $k = k - k_i$。

2.2 否则， $k_i = 0$。

2.3 $k = k / 2$, $i = i + 1$。

3、返回 $(k_{i-1}, k_{i-2}, \cdots, k_1, k_0)$。

举个简单的例子，正整数 k = 15，其二进制表示为 $k = (1111)_2$，通过算法 4 的计算可以得到 $NAF(k) = (1000\bar{1})_2$，其中 $\bar{1} = -1$。通过对比可以看出 NAF(k)比 k 的非零数字减少了。

算法 5 用二进制 NAF 方法计算点乘

输入：一个正整数 k，$P \in E(F_p)$.

输出：kP。

1、用算法 4 计算 NAF(k)。

2、Q = 0;

3、对于 i 从 m – 1 到 0，重复执行

3.1 Q = 2Q。

3.2 若 $k_i = 1$， 则 Q = Q + P。

3.3 若 $k_i = -1$， 则 Q = Q - P。

4、返回（Q）。

算法 5 [1]修改了从左到右的二进制点乘算法（算法 2），用 NAF(k)代替了 k 的二进制表示。根据 NAF 的性质可以得到算法 5 的期望运行时间近似为：

$$\frac{m}{3}A + mD$$

比算法 2 减少了 m/6 次的点加运算，当 m 的数值很大时，运算效率有所提高，代价是要增加算法 4 的运算。

二进制 NAF 方法是可以拓展的，注意在算法 4 中有一个运算步骤：2.1 若 k 为奇数，则 $k_i = 2 - (k \bmod 4)$, $k = k - k_i$，这一步对 k_i 的运算可以改成 $k_i = k \bmod 2^w$，

此时随着 w 的增加，算法 4 的运算难度将会增大，但是如果利用额外的存储器存储计算结果，算法 5 的运行时间还会进一步减少，在具体设计时可以折中考虑选取最佳的 w 值 [52], [53]。这种方法通常称作窗口 NAF 方法，w 称作窗口宽度。

2.6 椭圆曲线加密方案

椭圆曲线加密方案是任何一种椭圆曲线加密标准包括椭圆曲线数字签名算法（ECDSA）的核心。它是基于椭圆曲线上的离散对数问题（ECDLP）来制定的，不过完整的一个密码协议非常复杂，包括随机生成有限域椭圆曲线方程、加密方案参数组的生成、加密方案密钥对的生成、具体的签名算法和认证算法等几部分组成。本设计的主要目的是研制和开发适合于所有加密方案的运算芯片，并不涉及协议本身，因此下面只介绍加密方案密钥对的生成以及各种协议通用的加密方案这两部分。

2.6.1 椭圆曲线上的离散对数问题（ECDLP）

离散对数问题是很多密码体制的安全基础，为了更好的理解椭圆曲线加密方案，先来了解一下椭圆曲线离散对数问题（ECDLP：Elliptic Curve Discrete Logarithm Problem）。

椭圆曲线离散对数问题的具体定义如下 [1]：

给定定义于有限域 F_q（F_q 即可代表素数域 F_p 也可代表二进制域 F_{2^m} ）上的椭圆曲线 E，基点 $P \in E(F_q)$，阶为 n，点 $Q \in E(F_q)$，寻找一个正整数 $k \in [1, n-1]$，使得 $Q = kP$。正整数 k 称为 Q 的基于 P 的离散对数，表示为：

$$k = \log_p Q$$

应当指出 E 上不是任意两个点都存在离散对数解 k。

为了抵抗所有已知的对 ECDLP 的攻击，密码方案中使用的椭圆曲线的参数应该谨慎选择。最简单的求解 ECDLP 问题的算法是穷举搜索：连续计算一系列点 P，2P，3P，4P，…，直到得到 Q。该方法的运行时间最坏情况下为 n 步（n 为椭圆曲线上点 P 的阶），平均情况下为 n/2 步。因此可以通过选择参数 n 足够大的椭圆曲线，使得穷举攻击的计算量不可实际实现（如 $n \geq 2^{80}$）。对于 ECDLP，

已知最好的通用攻击方法是将 Pohlig-Hellman 算法 [54]和 Pollard's rho [55]算法相结合，该方法的运行时间为完全指数时间$O(\sqrt{p})$，其中 p 是 n 的最大因子。为了抵抗各种攻击，椭圆曲线参数的选择要求 n 含有大素数因子 p，p 应该足够大，以使得$\sqrt{p}$的计算量不可实际实现（如 $n > 2^{160}$）。另外，椭圆曲线的参数选择还要防止所有其他的已知攻击方法，这样所选择的 ECDLP 才会被认为对于当今的计算水平是难解的。

2.6.2 参数组

在给出椭圆曲线的加密方案之前一定要先给出椭圆曲线参数组的概念，它描述了一个具体的椭圆曲线，包括这个椭圆曲线 E 定义的有限域（素数域或二进制域），基点 $P \in E(F_q)$，还有阶 n。参数的选择要能使得 ECDLP 抵抗所有已知的攻击。这里还会出于安全性或者实现的原因而有其它的约束。典型地，参数组被一组实体共享，而在一些应用中有可能对每个用户特别设定。在本节中，我们将假定使用的是椭圆曲线所在的有限域为素数域 F_p 或者二进制域 F_{2^m} 。

通常参数组用大写字母 D 标示，参数组 D 中包含的参数为：

参数组 D = (q, FR, S, a, b, P, n, h)。

这些元素分别所代表的含义如下 [1]：

1、q 指有限域的阶。

2、FR 表示有限域，即素数域还是二进制域。

3、S 为椭圆曲线生成时的种子。一般情况下，为了保证加密方案的绝对安全，加密时所使用的椭圆曲线是随机生成的，有相关的算法来计算椭圆曲线方程中的参数，但是需要有一个种子，S 就是种子。

4、a, b 是椭圆曲线方程中的两个系数，而且有 $a, b \in F_q$。参数 a, b 定义了有限域 F_q 上椭圆曲线 E 的等式。（素数域中的椭圆曲线方程为：$y^2 = x^3 + ax + b$，二进制域中椭圆曲线方程为：$y^2 + xy = x^3 + ax^2 + b$）。

5、P 就是椭圆曲线上的基点。有限域 F_q 中的两个域元素 x_p 和 y_p，在仿射坐标中定义了一个有穷远点 $P = (x_p, y_p) \in E(F_q)$，P 有素数阶，因此称点 P 为椭圆

曲线上的基点。

6、n 表示基点 P 的阶。

7、h 为余因子，且满足等式 h = #E(F_q)/n。余因子 h 是对加密方案做验证时所使用的重要参数。

对于参数组 D = (q, FR, S, a, b, P, n, h)中的各个元素的选取要有约束，即密码体制的安全性约束。为了避免针对圆曲线离散对数问题（ECDLP）的 Pohlig-Hellman 算法 [54]和 Pollard's rho 算法 [55]的攻击，#E(F_q)必须能被足够大的素数 n 整除。最小应该保证 $n > 2^{160}$。给定域 F_q，为了最大限度地抵抗 Pohlig-Hellman 算法 [54]和 Pollard's rho 算法 [55]的攻击，应选择 E 使得#E(F_q)为素数或者近似素数，即#E(F_q) = hn，其中 n 为素数，h 非常小（如 h = 1，2，3 或 4）。

为了抵抗同构攻击（一种针对 ECC 的常用攻击算法），还要有更进一步的预防。为了避免攻击素数域异常曲线，应该保证#E(F_q)≠q。为了避免 Weil 和 Tate 配对攻击，应该保证对于所有的$1 \le k \le C$，n 不被 $q^k - 1$ 整除，其中 C 应该足够大，以使 F_q^*c 上的 DLP 不可解（若 $n > 2^{160}$，则 C = 20 就已足够大）。最后，为了避免 Weil 下降攻击，应该仅使用 m 为素数的 F_{2^m} 。

为了防止针对特殊类型椭圆曲线的攻击，应随机选择满足#E(F_q)能被大素数整除的椭圆曲线 E。因为随机曲线满足已知的同构攻击的条件的概率非常小而且可以忽略，因此已知攻击仍可避免。可以依照一些预先规定的过程将单向函数例如 SHA-1 的输出作为定义椭圆曲线的系数，而可证明的随机选择一个曲线。函数的输入种子 S 作为证据（在假定 SHA-1 单向性的前提下）证明椭圆曲线的确是随机生成的。这保证了椭圆曲线的用户不被通过故意构造有潜在弱点的曲线而窃取用户的私钥 [1]。

通过上面的简要分析我们知道参数组中的 8 个重要参数不是随意挑选的，而是要根据密码体制安全性的约束来选取。

下面的算法 6 [1, 56]单向生成密码学安全的参数组——上面讨论过的所有安全性约束都已满足。参数组可以通过算法 6 明确地加以确认。确认过程证明正在讨论的椭圆曲线有声明的阶，能抵抗针对 ECDLP 的所有已知攻击，并且基点 P 也有声明的阶。使用不可信的软硬件或第三方所获得的椭圆曲线同样可以使用该

算法来确认其密码学安全性。

在给出具体的参数组生成算法之前，先对参数组中出现的一些参数做出如下说明：

（1）因为选择 n 时要满足 $n > 2^L$，所以算法 6 的输入要求 $L \geq 160$，以确保 $n > 2^{160}$。

（2）条件 $L \leq [\log_2 q] + 1$ 确保 $2^L \leq q$，F_q 上的椭圆曲线 E 的阶#E(F_q)能被 L 位的素数整除（由于#E(F_q)≈q）。另外，若 $q = 2^m$，则 L 应满足 $L \leq [\log_2 q]$，因为二进制域的阶#E(F_{2^m})为偶数。

（3）条件 $n > 4\sqrt{q}$ 保证 E(F_q)有唯一的阶为 n 的子群，因为由 Hasse 定理有 #E(F_q)$\leq (\sqrt{q}+1)^2$，从而 n^2 不能够整除#E(F_q)。此外，由于 hn = #E(F_q)取决于 Hasse 区间，从而只有一个可能的 h 使得#E(F_q) = hn，即 $h = [(\sqrt{q}+1)^2 / n] + 1$。

在算法 6 中，候选椭圆曲线 E 会使用随机生成算法来生成椭圆曲线 E 的两个重要系数 a，b。对于素数域通过 SEA 点计数算法，对于二进制域通过 Satoh 的点计数算法得到阶#E(F_q)。对于 F_q 上的椭圆曲线 E 的阶#E(F_q)，若 F_q 是素数域，则该阶粗略地均匀分布于 Hasse 区间 $[q+1-2\sqrt{q}, q+1+2\sqrt{q}]$，若 F_q 是二进制域，则该阶粗略地均匀分布于 Hasse 区间中的偶数。因此在得到椭圆曲线的素数或近似素数阶以前，可以通过对 Hasse 区间中的每一个素数的估计来精确估算椭圆曲线的实际点数。对于候选曲线的测试，可以采用早期终止来加速，首先由 SEA 算法快速确定阶#E(F_q)模小素数 l，淘汰阶#E(F_q)能被 l 整除的曲线。只有通过这些测试的椭圆曲线才进行全部的点的计数算法。

使用随机曲线的一种替代方法是选择曲线的子域或者使用复乘方法 [57], [58]。算法 6 [1], [56]通过简单的修改后可提供这些选择的方法。

算法 6 生成参数组

输入：域的阶为 q，F_q 的域表示 FR，安全级别 L 满足 $160 \leq L \leq [\log_2 q] + 1$ 和 $2^L \geq 4\sqrt{q}$。

输出：参数组 D = (q, FR, S, a, b, P, n, h)。

1、若 F_q 是素数或二进制域，则对应地分别用不同的算法来随机生成系数 $a,b \in F_q$。令 S 为生成的种子。若 F_q 是素数，则令 E 为 $y^2 = x^3 + ax + b$；若 F_q 是二进制域，则令 E 为 $y^2 + xy = x^3 + ax^2 + b$。

2、计算 $N = \#E(F_q)$。

3、检验 N 是否能被满足 $n > 2^L$ 的大素数 n 整除。若不能，则跳至步骤 1 重新开始。

4、检验 n 对于所有 $1 \le k \le 20$ 是否能被 $q^k - 1$ 整除。若能，则跳至步骤 1 重新开始。

5、检验是否 $n \ne q$。若不是，则跳至步骤 1 重新开始。

6、令 h = N / n。

7、选择任意的点 $P' \in E(F_q)$ 并令 $P = hP'$。重复此运算一直到 $P \ne \infty$。

8、返回参数组 D = (q, FR, S, a, b, P, n, h)。

下面的算法 7 [1], [56]是对生成的参数组进行确认的算法。确认参数组的正确性对于使用加密方案的各方来说都是很重要的。

算法 7 参数组的确认

输入：参数组 D = (q, FR, S, a, b, P, n, h)。

输出：接受或拒绝参数组 D 的有效性。

1、检验 q 是否是素数的乘积（$q = p^m$，其中 p 为素数且 $m \ge 1$）。

2、若 p = 2，则检验 m 是否为素数。

3、检验 FR 是否是有效的域表示（通常情况下 FR 仅表示两种有限域：素数域和二进制域）。

4、检验 a，b，x_p，y_p（其中 P = (x, y)）是否是 F_q 上的元素（例如，检验它们是否符合 F_q 上元素的正确的格式）。

5、检验 a 和 b 是否定义 F_q 上的椭圆曲线（例如，对于素域要求 $4a^3 + 27b^2 \ne 0$，对于二进制域则要求 $b \ne 0$）。

6、若椭圆曲线是随机生成的，则

6.1 检验种子 S 是长度至少为 l 比特的二进制串，其中 l 为杂凑函数 H 的长度。

6.2 对素域和二进制域要通过各自的算法来检验 a 和 b 是否由种子 S 所生成。

7、检验是否 $P \neq \infty$。

8、检验 P 是否满足由 a 和 b 定义的椭圆曲线方程。

9、检验 n 是否是素数，$n > 2^{160}$，$n > 4\sqrt{q}$。

10、检验是否 $nP = \infty$。

11、计算 $h' = [(\sqrt{q}+1)^2 / n] + 1$，并检验是否 $h = h'$。

12、检验对于所有 $1 \le k \le 20$，n 是否不被 $q^k - 1$ 整除。

13、检验是否 $n \neq q$。

14、若任何一项检验失败，则返回（“无效”）；

否则，返回（“有效”）。

2.6.3 密钥对

在前面介绍椭圆曲线离散对数问题（ECDLP）时已经指出，在加密方案中对椭圆曲线参数的选择非常重要，它直接决定一个加密方案是否可行，在众多的参数中以公钥和私钥这一对密钥对最为重要。下面我们将详细介绍密钥对的生成及其确认。

椭圆曲线密钥对与参数组 D = (q, FR, S, a, b, P, n, h)中的一系列参数有关。在由点 P 生成的群上随机选择点 Q 作为公钥。相应的私钥是 $d = \log_P Q$。要生成密钥对的实体 A 必须保证参数组是合法的。参数组和公钥间的关系必须能被所有随后可能用到 A 的公钥的通信实体所检验。在实践中这种关系可由密码学途径（例如一个认证中心生成证明该关系的证书）或书面（如所有实体使用同样的参数组）达成。算法 8 [1], [59], [60]是密钥对生成算法。

从公钥 Q 计算私钥 d 的问题显然就是椭圆曲线离散对数问题。因此至关重

要的是参数组 D 的选择要使得 ECDLP 不可求解。此外数 d 的生成要是“随机”的，以防止攻击者给予概率搜索策略获得额外信息。

算法 8 密钥对的生成

输入：参数组 D = (q, FR, S, a, b, P, n, h)。

输出：公钥 Q，私钥 d。

1、选则 d∈[1, n－1]。

2、计算 Q = dP。

3、返回 (Q, d)。

确认公钥的目的是检验公钥拥有可靠的算术性质。执行成功表明对应的私钥逻辑上存在，尽管不能表明某人已经计算出私钥，也不能表明主体确实用于它。公钥确认在基于 Diffie-Hellman 的密钥建立协议中特别重要，实体 A 将自己的私钥和从另一个实体 B 处收到的公钥相结合而得出共享的秘密 k，随后使用 k 进行对称密码协议（如加密或者消息认证）。一个不诚实的 B 会选择一个非法的公钥通过上述途径使用 k 窃取 A 的私钥的信息。

检验 nQ = O 有比对 nQ 执行点乘更快的方法。例如当 h = 1（这是实际使用素数域上椭圆曲线的常见情况）时，算法 9 [1], [59], [60]中的步骤 1、步骤 2 和步骤 3 可直接推出 nQ = O。在一些协议中对 nQ = O 的检验可以省略，或者嵌入到协议的计算中，或者由对 hQ ≠ O 的检验所取代。后者的检验可确保 Q 不在阶能被 h 整除的椭圆曲线的小子群上。具体的检验可参看下面给出的算法 10 [1], [59], [60]。

算法 9 公钥确认

输入：参数组 D = (q, FR, S, a, b, P, n, h)，公钥 Q。

输出：判断 Q 是否合法。

1、检验是否 Q ≠ O。

2、检验 x_Q，y_Q 是否是有限域 F_p 或 F_{2^m} 上元素的正确表示（例如，若是素域 F_p，则应在整数区间[0, p - 1]，若是二进制域 F_{2^m}，则数串的长度应为 m 比特）。

3、检验 Q 是否满足由 a 和 b 定义的椭圆曲线。

4、检验是否 nQ = O（无穷远点）。

5、若任何检验失败，则返回（“拒绝”）；否则，返回（“接受”）。

算法 10 嵌入的公钥确认

输入：参数组 D = (q, FR, S, a, b, P, n, h)，公钥 Q。

输出：判断 Q 是否（部分）合法。

1、检验是否 $Q \neq O$。

2、检验 x_Q，y_Q 是否是有限域 F_p 或 F_{2^m} 上元素的正确表示（例如，若是素域 F_p，则应在整数区间[0, p - 1]，若是二进制域 F_{2^m}，则数串的长度应为 m 比特）。

3、检验 Q 是否满足由 a 和 b 定义的椭圆曲线。

4、若任何检验失败，则返回（“拒绝”）；否则，返回（“接受”）。

2.6.4 加密方案

在介绍完椭圆曲线加密的密钥对如何生成以及确认以后我们给出具体的加密方案，下面将要给出的加密方案是具有普遍性的，它是任何加密协议的基础。

算法 11 基本椭圆曲线加密

输入：椭圆曲线参数组 D = (q, E, P, n)，公钥 Q，明文 m。

输出：密文 (C_1, C_2)。

1、将明文 m 表示成 $E(F_q)$上的点 M。

2、计算 $C_1 = kP$。

3、计算 $C_2 = M + kQ$。

4、返回 (C_1, C_2)。

算法 12 基本椭圆曲线解密

输入：椭圆曲线参数组 D = (q, E, P, n)，私钥 d，密文 (C_1, C_2)。

输出：明文 m。

1、计算 $M = C_2 - dC_1$，并从点 M 中取出明文 m。

2、返回 (m)。

在算法 11 [1]和算法 12 [1]中，椭圆曲线参数组 D = (q, E, P, n)中的 q 为有限域的阶，素数域中为素数 p，二进制域中为二进制 2^m。E 为有限域上的椭圆曲线，P 是椭圆曲线上的一点，n 为点 P 在椭圆曲线上的阶。

首先把明文 m 表示成一个点 M，然后再加上 kQ 进行加密，其中 k 是随机选择的正整数，Q 是接收方的公开密钥。发送方把密文 $C_1 = kP$ 和 $C_2 = M + kQ$ 发给接收方。接收方用自己的私钥 d 计算：

$$dC_1 = d(kP) = k(dP) = kQ$$

进而可恢复出明文 $M = C_2 - dC_1$。攻击者若想恢复出明文 M，则需要计算 kQ。而从公开参数组，Q 和 $C_1 = kP$ 计算 kQ 则是椭圆曲线离散对数问题（ECDLP）。

2.6.5 椭圆曲线数字签名算法

在前面已经详细介绍了椭圆曲线的加密方案，下面将介绍这种方案的一种最为普遍的应用，椭圆曲线数字签名算法（ECDSA） [61]，它可以算是数字签名算法（DSA）的椭圆曲线版本。最广泛标准化的基于椭圆曲线的签名方案包括 ANSI X9.62、FIPS 186-2、IEEE 1363-2000 和 ISO/IEC 15946-2 标准，以及一些标准的草案 [1]。

算法 13 ECDSA 签名的生成

输入：参数组 D = (q, FR, S, a, b, P, n, h)，私钥 d，消息 m。

输出：签名 (r, s)。

1、选择 $k \in [1, n-1]$。

2、计算 $kP = (x_1, y_1)$并将 x_1 转换为整数 $\overline{x_1}$。

3、计算 r = $\overline{x_1}$ mod n。若 r = 0，则跳至步骤 1。

4、计算 e = H(m)。

5、计算 $s = k^{-1} (e + dr) \bmod n$。若 s = 0，则跳至步骤 1。

6、返回 (r, s)。

算法 14 ECDSA 签名的验证

输入：参数组 D = (q, FR, S, a, b, P, n, h)，公钥 Q，消息 m，签名(r, s)。

输出：判断签名是否合法。

1、检验 r 和 s 是否是区间[1, n - 1]内的整数。若任何一个检验失败，则返回（“拒绝该签名”）。

2、计算 e = H(m)。

3、计算 $w = s^{-1} \bmod n$。

4、计算 $u_1 = ew \bmod n$ 和 $u_2 = rw \bmod n$。

5、计算 $X = u_1P + u_2Q$。

6、若 X = O（无穷远点），则返回（“拒绝该签名”）。

7、将 X 的 x 坐标 x_1 转换为整数 $\overline{x_1}$；计算 v = $\overline{x_1}$ mod n。

8、若 v = r，则返回（“接受该签名”）；否则，返回（“拒绝该签名”）。

在算法 13 [1], [61]和算法 14 [1], [61]中 H 表示一个密码杂凑函数，其输出长度不超过 n 比特（若该条件不满足，则可将 H 的输出截断）。

下面给出算法 14 的简要证明过程 [1]：

若一条信息 m 的签名(r, s)确实是由合法的签名者所生成的，则有 $s = k^{-1} (e + dr) \bmod n$，重新整理可得：

$k = s^{-1} (e + dr) \bmod n$

$= (s^{-1} e + s^{-1} rd) \bmod n$ （因为 $w = s^{-1} \bmod n$）

$= (we + wrd) \bmod n$ （因为 $u_1 = ew \bmod n$ 和 $u_2 = rw \bmod n$）

$= (u_1 + u_2 d) \bmod n$

再有：$X = u_1P + u_2Q$ （因为 Q = kP）

$= (u_1 + u_2 d)P$

$= kP$

因此得证 $v = r$。

2.7 椭圆曲线密码体制的应用

椭圆曲线密码体制（ECC）的研究历史较短，但是由于优点突出，已经得到了密码界的重视并广泛应用。其是目前已知的所有公钥密码体制中能够提供最高比特强度（strength-per-bit）的一种公钥密码体制。在本节中除了介绍该算法的优点以外，还会给出椭圆曲线密码体制在应用过程中所涉及到的相关概念，例如在应用过程中所采用的统一标准有哪些、具体的应用领域是什么以及在应用过程中椭圆曲线加密算法的安全性问题。

2.7.1 椭圆曲线密码体制的优点

我们将椭圆曲线加密算法（ECC）与目前应用最为广泛的公钥密码体制 RSA 算法进行比较，椭圆曲线加密算法在以下 4 个方面具有明显的优势 [1], [42], [44]：

1、安全性能更高

加密算法的安全性能一般通过该算法的抗攻击强度来反映。ECC 和其他另外几种公钥系统相比，其抗攻击性具有绝对的优势。椭圆曲线离散对数问题（ECDLP）计算困难性在计算复杂度上目前并没有亚指数级的通用方法，即它的计算复杂度为指数级的，而 RSA 是亚指数级的。这体现为 ECC 比 RSA 的每 bit 安全性能更高。

2、计算量小和处理速度快

在相同的计算资源条件下，虽然在 RSA 中可以通过选取较小的公钥（可以小到 3）的方法提高公钥处理速度，即提高加密和签名验证的速度，使其在加密和签名验证速度上与 ECC 有可比性，但是私钥的处理速度上（解密和签名），ECC 远比 RSA、DSA 快得多。因此 ECC 总的速度比 RSA、DSA 要快得多。同时 ECC 系统的密钥生成速度比 RSA 快百倍以上。因此在相同条件下，ECC 则有更高的加密性能。

3、存储空间占用小

ECC 的密钥尺寸和系统参数与 RSA、DSA 相比要小得多。160 位 ECC 与 1024 位 RSA、DSA 具有相同的安全性能，210 位 ECC 则与 2048 位 RSA、DSA 具有相同的安全强度。意味着它所占的存储空间要小得多。这对于加密算法在资源受限制的环境下（如智能卡等）的应用具有特别重要的意义。

4、带宽要求低

当对长消息进行加解密时，ECC、RSA、DSA 三类密码系统有相同的带宽要求，但应用于短消息时 ECC 带宽要求却低得多。而公钥加密系统多用于短消息，例如用于数字签名和用于对称系统的会话密钥加密传递。带宽要求低使得 ECC 在无线网络领域具有更加广泛的应用前景。

2.7.2 椭圆曲线密码体制的相关标准

ECC 的上述优点使它在某些领域（PDA、手机、智能卡）的应用将可能取代 RSA，并成为通用的公钥加密算法。许多国际标准化组织（政府、工业界、金融界、商业界等）已将各种椭圆曲线密码体制作为其标准化文件向全球颁布。

ECC 标准大体可以分为两种形式 [1], [42], [44]。

一类是技术标准，即描述以技术支撑为主的 ECC 体制，主要有 IEEE P1363、ANSI X9.62、ANSI X9.63、SEC1、SEC2、FIP186-2、ISO/IEC14888-3 等。规范了 ECC 的各种参数的选择，并给出了各级安全强度下的一组 ECC 参数。当然这些标准也包括了 ECDSA 的标准；其中有 1998 年为 ISO 颁布，1999 年 ANSI、2000 年 IEEE 和 NIST 颁布；另外还有一个有影响的组织 SECG(Standards for Efficient Cryptography)也颁布了 ECDSA 标准。

另一类是应用标准，即在具体的应用环境中建议使用 ECC 技术，主要有 ISO/IEC15946、IETF PKIX、IETF TLS、WAP WTLS 等。

在标准化的同时，一些基于标准（或草案）的各种椭圆曲线加密、签名、密钥交换的软、硬件也相继问世。可以相信，ECC 技术在信息技术安全领域中的应用会越来越广。

在各个标准下都推荐了一些常用的椭圆曲线，在保证了安全性的同时进一步规范了 ECC 的应用。被推荐的椭圆曲线可以分为三类 [1], [42], [44]：

（1） 基于 F_p 的随机产生的椭圆曲线。

（2） 基于 F_{2^m} 的随机产生的椭圆曲线。

（3） 基于 F_{2^m} 的 Koblitz 椭圆曲线。

下面给出几个具体的实例，这些椭圆曲线的参数是 NIST 给美国联邦政府使用推荐的：

1、基于素域 F_p 的有：

$p = 2^{192} - 2^{64} - 1$

$p = 2^{224} - 2^{96} + 1$

$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

……

2、F_{2^m} 的有：$F_{2^{163}}$，$F_{2^{233}}$，$F_{2^{283}}$，$F_{2^{409}}$，$F_{2^{571}}$，……

2.7.3 椭圆曲线密码的适用领域

由于 ECC 实现较高的安全性，只需要较小的开销和时延，较小的开销体现在如计算量、存储量、带宽、软硬件实现的规模等，时延体现在加密或签名认证的速度等方面。所以 ECC 特别适用于计算能力和集成电路空间受限（如智能 IC 卡）、带宽受限（如无限通信和某些计算机网络）、要求高速实现的情况。

可以列举如下 [42], [44]：

1、无线 Modem 的实现。对分组交换数据网提供数据加密，例如在移动通信器件上运行 4MHz 的 68330CPU，ECC 可以实现快速的 Diffie-Hellman 密钥交换，并以极小化密钥交换占用宽带。

2、用于 Web 服务器。在 Web 服务器上集中进行密码运算会形成瓶颈，Web 服务器的带宽有限并且带宽的费用高昂，采用 ECC 可以节省计算时间和带宽。

3、集成电路卡的实现。ECC 不需要协处理器就可以在标准卡上实现快速、安全的数字签名，这是 RSA 等其它密码体制难以实现的。ECC 可以使程序代码、密钥、证书的存储空间极小化，便于实现，大大降低了 IC 卡的成本。

自从 1985 年 N. Koblitz 和 Miller 提出将椭圆曲线用于密码算法以来，椭圆曲线密码体制得到了很大的发展，已经成为密码学的重要研究热点之一。由于椭圆曲线体制的启发，还有人提出了其它类型的曲线，如超椭圆曲线和圆锥曲线等，

并提出了在这些曲线上实现公钥密码算法等观点。

2.7.4 椭圆曲线密码的安全性

在评估一个密码协议的安全性时，通常假设敌手知道协议的全部细节，掌握所有的公钥，只是缺少私钥的信息。此外敌手能够截取合法通信方之间传递的一些数据，甚至可以对这些数据进行控制（例如对于一个签名方案通过选择消息进行选择明文攻击，或者对于一个公钥加密方案通过选择密文进行选择密文攻击）。敌手试图通过求解被假定困难的问题，或者寻找协议设计中的一些缺陷，来危及协议的安全。

在这种传统安全模型中考虑的攻击都是从协议规范的数学基础来进行分析。近年来，研究者越来越注意到可能利用实现和操作环境的特性的攻击。这样的旁门攻击利用协议执行时信息泄漏，这在传统的安全模型中没有考虑到 [62], [63], [64]。例如当使用智能卡执行诸如解密和签名的生成等私钥操作时，敌手可能监视能量消耗或者电磁辐射。敌手还能够测量执行一个密码操作的时间，或者分析当遇到特定错误时密码设备的举动。在实践中很容易收集旁门信息，因此在全面评估系统的安全性时确定旁门攻击的威胁是至关重要的。

需要注意的是，在一些环境中特定的旁门攻击并不构成现实的威胁。例如，当密码设施是从外部不可信的来源处获取能量的智能卡时，对于密码设施测量能量消耗的攻击似乎是可行的。然而，若密码设施是位于一个安全的办公室中的工作站，则能量攻击就变得没什么威胁了。

这样看来，要想具体的评估椭圆曲线加密算法的安全性是非常困难的，因为它不仅与算法本身的数学模型有关，而且与其在实际应用中的环境联系也很紧密。椭圆曲线加密算法本身的数学模型保证这个算法的安全性是非常高的，因此这部分只给出几种常见的旁门攻击方法以及相应的防范对策。

1、能量分析攻击 [30], [65]。

2、电磁分析攻击 [30], [66], [67]。

3、错误分析攻击 [30], [68]。

5、时间分析攻击 [30], [69]。

对于上述 4 种常见旁门攻击的相应对策主要包括算法的、基于软件的、基于

硬件的或以上方法的组合。没有哪个对策能够保证防御所有的旁门攻击。此外，需要指出有些对策可能会降低密码的计算速度，需要消耗更多的存储或者硬件资源。

2.8 本章小结

这一章我们全面的介绍了椭圆曲线加密算法的相关知识，从椭圆曲线的概念到椭圆曲线上的基本运算，从基本运算到密码实现方案，最后从密码实现方案到密码体制的应用。所有的内容都是在我们研究范围之内，但是我们主要的工作是设计出一款可以高效的椭圆曲线加/解密运算芯片，这个运算芯片的核心运算就是椭圆曲线上的点乘。在本章的 2.5 节我们已经对于点乘的整体运算做出了非常重要的改进，从改进后的算法 3 可以清楚的看到点乘运算要分解成点加和倍点运算，在 2.4 节已经给出了点加和倍点运算的相关公式，从这些公式可以看出点加和倍点运算又要分解成有限域上的加、减、乘、除等有限域运算。下一章我们将详细讲述有限域运算。

第三章　有限域算术运算

通过在上一章对椭圆曲线的整体介绍我们可以看出整个加密方案的运算都是在有限域 F_p 或 F_{2^m} 上进行的，因此有限域算术运算的有效实现是椭圆曲线加密运算的重要先决条件，这也是本章和全文讨论的重点，在本章除了简单介绍有限域的理论以外，主要介绍在素数域 F_p 和二进制域 F_{2^m} 上的加、减、乘、求逆运算的算法及其硬件实现。本设计对椭圆曲线上的有限域算术运算做出了重要的改进，使得运算在算法和硬件实现上的效率更高。

3.1 节主要讲述有限域上的相关概念。3.2 节将详细讲述有限域上的加法和减法这两种运算。3.3 节将讲述有限域上的乘法运算和约简运算。3.4 节讲述有限域上的平方运算。3.5 节介绍有限域上最为复杂的运算——求逆运算。在总结 3.2 节到 3.5 节的基础上，3.6 节将讲述支持素数域和二进制域上任何运算的双域混合算法。3.7 节是本章小结，总结对算法的改进工作。

3.1 有限域简介

域是对常见的数系（如有理数、实数和复数）及其基本特性的抽象。域由一个集合 F 和两种运算共同组成，这两种运算分别为加法和乘法，这里简单的把加法和乘法所应满足的三个算术特性介绍一下 [44], [47]：

（1）集合 F 关于加法构成 Abel 群。该加法群单位元写为 0。

（2）集合 F 中除去元素 0 外，对乘法构成 Abel 群。该乘法的单位元写为 1。

（3）加法和乘法间有分配律：

$$a(b+c)=ab+ac$$
$$(b+c)a=ba+ca$$

若集合 F 是有限集合，即 F 中的元素个数有限，此时称这个域为有限域，前面提到的素数域 F_p 和二进制域 F_{2^m} 都是有限域。

3.1.1 域运算

这部分介绍一下域中的 4 种运算：加、减、乘、求逆。在这 4 种运算中，加法和乘法是在域的定义中已经有的。域元素的减法用加法来定义：对于 $a, b \in F$，$a - b = a + (-b)$，其中-b 是 b 的负元素，它是使得 $b + (-b) = 0$ 的唯一的一个域元素。类似地，域元素的除法也可以用乘法来定义：对于 $a, b \in F$，$b \neq 0$，$a / b = a \cdot b^{-1}$，其中 b^{-1} 是 b 的逆元素，它是使得 $b \cdot b^{-1} = 1$ 的唯一的一个域元素，通常把求解 b^{-1} 的运算称为求逆运算 [1], [47]。

3.1.2 素数域

在给出素数域的概念之前再一次给出有限域的阶的概念，有限域的元素的个数称为有限域的阶。当且仅当 q 是一个素数的幂，即 $q = p^m$，其中 p 是一个素数，m 是一个正整数时，存在一个 q 阶有限域 F，此时把 p 称为有限域 F 的特征。若 $m = 1$，则域 F 就称为素数域。若 $m \geq 2$，则域 F 为扩域。对于任意一个素数的幂 q，本质上只存在一个 q 阶有限域。这意味着，任意两个 q 阶有限域除了域元素的表示符号不同外，在结构上它们是相同的。我们称任意两个 q 阶有限域都同构，并用符号 F_q 表示它们。即 q 阶有限域满足存在性和唯一性两个性质，明确这一点对后面的研究很重要。

设 p 是一个素数，以 p 为模，则模 p 的全体余数的集合{0, 1, 2, …, p – 1}关于模 p 的加法和乘法构成一个 p 阶有限域，因为 $m = 1$，所以这个有限域称为素数域或素域，并用符号 F_p 表示，并且称 p 为 F_p 的模。对于任意整数 a，a mod p 表示用 p 除 a 所得到的余数 r，$0 \leq r \leq p - 1$，并称这种运算为求模 p 的剩余 [1], [47]。

下面举例说明素数域 F_{29} 上的加、减、乘、求逆四种运算：

加法：$17 + 20 = 37 \bmod 29 = 8$

减法：$17 - 20 = -3 \bmod 29 = 26$

乘法：$17 \times 20 = 340 \bmod 29 = 21$

求逆：$17^{-1} \bmod 29 = 12$

由于素数域 F_{29} 的元素只有 29 个，它们是{0, 1, 2, …, 28}，因此加、减、乘、求逆四种运算的结果都应该在这 29 个元素中，保证这一结果的关键是 a mod p

这个运算，即对 a 求模 p 的剩余，简称为对 a 取模。这也是有限域上的运算和普通运算最大的不同。

3.1.3 二进制域

阶为 2^m 的有限域称为二进制域或特征为 2 的有限域。构成 F_{2^m} 的一种方法是采用多项式基表示法。在这种表示方法中，二进制域 F_{2^m} 的元素是次数最高为 m – 1 次的二进制多项式，且这些多项式的系数为 $F_2 = \{0, 1\}$：

$$F_{2^m} = \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_2x^2 + a_1x + a_0 \qquad a_i \in \{0,1\}\}$$

同时还需要选择一个二进制域上的 m 次既约多项式 f(x)（对于任意的 m，这样的多项式一定存在，并且能有效产生）做为模。f(x)的既约性意味着 f(x)不能被分解成次数低于 m 的二进制域上的多项式的乘积。域元素的加法是两个多项式系数的模 2 相加。域元素的乘法是两个多项式相乘后取模 f(x)。对于任意一个二进制域上的多项式 a(x)，a(x) mod f(x)表示一个次数低于 m 的余式多项式 r(x)。余式多项式 r(x)可用 f(x)辗转相除 a(x)得到，这一运算通常称为求模 f(x)的余式 [1], [47]。

下面给出二进制域 F_{2^4} 上的算术运算的简单例子，在二进制域 F_{2^4} 上的约简多项式为 $f(x) = x^4 + x + 1$，也可简单表示成 $f(x) = 10011$，则

加法：$1101 + 0111 = 1010$

减法：$1101 - 0111 = 1010$

乘法：$1101 \times 0111 = 100011 \bmod 10011 = 0101$

求逆：$1101^{-1} \bmod 10011 = 0100$

3.1.4 扩域

二进制域的多项式基表示法可以按如下方法推广到所有的扩域。设 p 是一个素数，$m \geq 2$。令 $F_p[x]$表示所有系数取自 F_p 上的 x 的多项式集合。另 f(x)是 $F_p[x]$上的一个 m 次既约多项式，以 f(x)作为约减多项式。对于任意的 p 和 m，这样的约减多项式一定存在，而且能够有效地产生。f(x)的既约性意味着它不能分解为

F_p[x]上次数小于 m 的多项式的乘积。域 F_{p^m} 上的元素都是域 F_p[x]上次数小于 m 的多项式：

$$F_{p^m} = \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_2x^2 + a_1x + a_0 \qquad a_i \in F_p\}$$

有限域上的加法就是多项式的系数按照 F_p 的运算规则相加。域的乘法就是模 f(x)的多项式乘 [1], [47]。

下面给出一个扩域运算的简单例子。设 p = 251，m = 5。多项式 $f(x) = x^5 + x^4 + 12x^3 + 9x^2 + 7$ 是 F_{251}[x]上的既约多项式，以 f(x)作为约减多项式构造的域 F_{251^5}。F_{251^5} 的阶为 251^5，其元素都是 F_{251}[x]上的次数小于等于 4 的多项式。

令 $a = 123x^4 + 76x^2 + 7x + 4$，$b = 196x^4 + 12x^3 + 225x^2 + 76$，则在扩域 F_{251^5} 上的算术运算如下：

加法：$a + b = 68x^4 + 12x^3 + 50x^2 + 7x + 80$

减法：$a - b = 178x^4 + 239x^3 + 102x^2 + 7x + 179$

乘法：$a \times b = 117x^4 + 151x^3 + 117x^2 + 182x + 217$

求逆：$a^{-1} = 109x^4 + 111x^3 + 250x^2 + 98x + 85$

3.1.5 有限域的相关概念

在这一部分给出几个有限域常用的概念，方便后面解释有限域上的算术运算。在前面刚刚讨论完扩域的概念，这里首先给出与扩域相对应的一个概念 [1]：子域。设 k 是域 K 的子集合，且 k 对于 K 中的运算构成域，则称 k 为 K 的子域，称 K 是 k 的扩域。有限域的子域容易表征。对于 m 的每一个正因子 l，有限域 F_{p^m} 恰好有阶为 p^l 的子域，且子域的元素都是域 F_{p^m} 上满足 $a^{p^l} = a$ 的元素。反过来说，域 F_{p^m} 中的每一个子域的阶都是 p^l，l 是 m 的正因子。

在前面也曾经提到过有限域的基的概念 [1]。有限域 F_{q^n} 可以看做是 F_q 上的

一个向量空间。其向量是域 F_{q^n} 的元素，标量是域 F_q 上的元素，向量的加法运算是域 F_{q^n} 上的加法运算，标量与向量的乘法是域 F_q 上的元素与域 F_{q^n} 上的元素在域 F_{q^n} 上的乘法运算。这个向量空间的维数是 n，并有许多基底，我们把它们简称为基。

若 $B = \{b_1, b_2, \ldots, b_n\}$是一组基，则域 F_{q^n} 上的元素 a 可以唯一地表示为域 F_q 上的一个 n 维向量（$a_1, a_2, \ldots, a_n$），使得：

$$a = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

例如，在域 F_{p^m} 上的多项式基表示中，域 F_{p^m} 是域 F_p 上的一个 m 维向量空间，并且$\{x^{m-1}, x^{m-2}, \ldots, x^2, x, 1\}$是一组基。

最后再给出有限域乘法群的概念 [1]。有限域 F_q 上的全体非零元素构成一个乘法循环群，记为 F_q^*。F_q^*中存在一个元素 b，b 称为生成元，使得

$$F_q^* = \{b^i : 0 \le i \le q-2\}$$

$a \in F_q^*$的阶满足 $a^t = 1$ 的正整数 t。因为 F_q^*是循环群，所有 t 是 $q-1$ 的因子。循环群的概念是很多有限域求逆算法的基础，在后面介绍有限域求逆算法时还会提到这个重要的概念。

3.2 有限域加减法

从这部分开始我们将详细讨论椭圆曲线加密算法的两种常用的有限域——素数域 F_p 和二进制域 F_{2^m} 上的加法、减法、乘法和求逆这四种主要的算法运算。并在每一部分的讨论中都把两种不同有限域的运算统一起来，给出最为适合硬件实现的统一算法。先介绍有限域加法和减法。

3.2.1 素数域加减法

在前面给出的素域 F_p 中加减法的具体实例可以看出，素域上的加减法和通常的代数学中的整数加减法基本相同，唯一的不同是由于计算结果必须保证是素域 F_p 中的元素，所以对最后的结果要增加一步“求模 p 的剩余”的运算，就是

我们通常所说的取模运算 [51], [70]。

下面给出素域 F_p 上加法和减法的具体算法：

算法 15 素数域 F_p 上的加法

输入：模数 p 和整数 a, b∈[0, p - 1]。

输出：c = (a + b) mod p。

1、c = a + b。

2、若 c ≥ p，则 c = c – p。

3、返回（c）。

算法 16 素数域 F_p 上的减法

输入：模数 p 和整数 a, b∈[0, p - 1]。

输出：c = (a – b) mod p。

1、c = a – b。

2、若 c ≤ 0，则 c = c + p。

3、返回（c）。

上述两个算法非常简单地描述了素数域 F_p 上的加减法，分别在算法 15 的步骤 2 和算法 16 的步骤 2 完成了对计算结果的取模运算。素域 F_p 上的加减法最多需要完成一个加法运算、一个减法运算和一个比较运算。应当强调的是在我们用硬件来实现这两个算法时并没有直接执行算法中的比较运算，具体的执行方法在硬件实现那一章将做出详细说明。

3.2.2 二进制域加减法

二进制域 F_{2^m} 上的算术运算和素数域以及代数学中的整数运算都有较大的区别，这是由于二进制域中元素的表示方式不同造成的。在密码学中，二进制域中的元素多采用多项式基表示法。在这种表示方法中，二进制域 F_{2^m} 的元素是次

数最多为 m – 1 次的二进制多项式，且这些多项式的系数为 F_2 = {0, 1}。因此通常用一个位数最多为 m – 1 位的二进制数来表示一个元素，它的每一位对应于多项式的每一个系数。这种表示法使得素数域中的加法和减法对应于二进制域中的按位异或运算“⊕”。素数域中的乘以 2 和除以 2 的运算分别对应于二进制域中的左移运算“<<”和右移运算“>>”。

算法 17 二进制域 F_{2^m} 的加减法

输入：次数低于 m 的二进制多项式 a(x), b(x)。

输出：c(x) = a(x) + b(x)。

1、C[i] = A[i] ⊕ B[i]。

2、返回（c(x)）。

在算法 17 中 a(x), b(x)和 c(x)表示二进制域中的元素，它们为三个多项式，A(x), B(x)和 C(x)是 a(x), b(x)和 c(x)对应的三个二进制数。算法 17 中并没有取模的运算，因为在二进制域 F_{2^m} 中作为模的既约多项式 f(x)是次数为 m 次的多项式，而次数低于 m 的二进制多项式 a(x), b(x)的加法运算结果并没有溢出，因此并不需要进行取模运算。另外，在二制域的多项式表示法中由于多项式的系数为 F_2 = {0, 1}，因此加法和减法的结果一样，同为异或“⊕”运算后的结果。这样二进制域 F_{2^m} 上的加减法只需要完成一个按位异或运算即可。

3.2.3 素数域和二进制域加减法比较

总体来说，素数域和二进制域的加减法的运算难度都不大。通过比较算法 15、算法 16 和算法 17 可以看出二进制域的加减法非常简单，只是在做按位异或运算，不但省去了每位之间的进位运算，最后的结果也不用进行取模运算，因此对于二进制域加减法来说，运算时间是固定的，并不随着操作数位数的增加而增加。相比之下，素数域加减法的运算不但要完成整数加减法的运算，而且还要通过比较运算来进行条件判断，输入数据的变化将会影响到计算时间的长短。

3.3 有限域乘法

无论是素数域 F_p 还是二进制域 F_{2^m}，乘法的复杂性都较加减法要高很多，主要原因是由于有限域上的两个元素的乘积将会变得很大，远远超出模的范围，因此要经过很多次的取模运算才能得到最后的结果，怎样使得最后的算法在做乘法运算和取模运算时达到最高的效率将是我们这节讨论的重点。

3.3.1 素数域乘法

素数域 F_p 上的乘法 $a \cdot b$ 可以通过先将 a 和 b 作整数乘法，然后以 p 为模做取模运算来完成。算法 18 [1]和算法 19 [1]分别给出了两种非常通用的整数乘法算法，它们分别阐明了基本运算数扫描和乘积扫描的方法。取模的算法将在后面单独做出介绍。

整数乘法算法

素数域 F_p 的元素是从 0 到 p – 1 的整数。令

$$m = [\log_2 p] + 1$$

是 p 的二进制长度，t = [m / W] + 1 是表示该域元素所用的 W 位字的个数（在后面的算法中如果不加说明，素数域的元素都满足以上条件）。这样素数域的一个元素 a 可以表示成一个二进制形式的数存储在一个 t 个 W 位字的数组 A 中：

$$A = (A[t-1], \cdots A[2], A[1], A[0])$$

$$a = 2^{(t-1)W} A[t-1] + \cdots + 2^{2W} A[2] + 2^{W} A[1] + A[0]$$

其中最后边的 A[0]是最低有效位，如表 3.1。

表 3.1　素域元素 a 的二进制表示

A[t – 1]	…	A[2]	A[1]	A[0]

在算法 18 和 19 中，(UV)表示一个（2W）位的量，这个量是 W 位的字 U

和 V 的级联。

算法 18 整数乘（运算数扫描方式）

输入：整数 a，b∈[0, p - 1]。

输出：c = a · b。

1、对于 $0 \le i \le t-1$，令 C[i] = 0。

2、对于 i 从 0 到 t – 1，重复执行

2.1 U = 0。

2.2 对于 j 从 0 到 t – 1，重复执行

(UV) = C[i+j] + A[i] · B[j] + U。

C[i+j] = V。

2.3 C[i + t] = U。

3、返回（c）。

在算法 18 中的步骤 2.2 计算 C[i+j] + A[i] * B[j] + U 的运算称为求中间积。因为运算数是 W 位的数，所以乘积的上界为：

$$2(2^W-1)+(2^W-1)^2=2^{2W}-1$$

并没有超出 2^{2W}，即可以用一个 2W 位的数表示，因此能用(UV)表示。如果 W = 1，即把整数 a 和 b 都表示成二进制数，则算法 18 实际完成的就是两个二进制数乘法运算。通过对字长 W 的调整，可以使算法 18 的效率最好。

算法 19 的编排使$c=a\cdot b$的计算从右向左进行。与前一个算法一样，两个 W 位的数相乘，要产生一个（2W）位的乘积。在算法 19 中，R_0，R_1，R_2，U 和 V 都是 W 位的字。

算法 19 整数乘（积扫描方式）

输入：整数 a，b∈[0, p - 1]。

输出：c = a · b。

1、$R_0 = 0$，$R_1 = 0$，$R_2 = 0$。

2、对于 k 从 0 到 2t – 2，重复执行 3

2.1 对于集合{(i, j) | i + j = k, 0 ≤ i, j ≤ t -1}中的每一个元素，重复执行

(UV) = A[i] · B[i]。

(ε , R_0) = R_0 + V。

(ε , R_1) = R_1 + U + ε 。

R_2 = R_2 + ε 。

2.2 C[k] = R_0，R_0 = R_1，R_1 = R_2，R_2 = 0。

3、C[2t - 1] = R_0。

4．返回（c）。

算法 19 [1]和算法 18 [1]大体相似，都是用求中间积的方法来实现最后的运算。如果直接运算 a 和 b，由于 a 和 b 是两个较大的乘数，因此这个运算需要耗费大量的资源，而且要花很长时间才能完成。现在把 a 和 b 这两个数都分割成由 t 个 W 位字长的数组成，先求出中间积，再通过叠加和移位来完成最后的乘积运算。这种方法效率更高，因此在大数的整数乘法中广泛应用。由于算法 19 是先计算中间积，然后对中间积进行移位和加法运算，因此增加了中间变量来存放中间结果，但是算法 19 只用一个单循环替代了算法 18 中的循环嵌套。因此可以说这两个算法各有千秋。

Karatsuba-Ofman 乘法算法

对于两个 n 位的整数乘法来说，算法 18 [1]和算法 19 [1]需要 $O(n^2)$次为运算。一种由 Karatsuba 和 Ofman 提出的“分而治之”算法将计算复杂性减少到了 $O(n^{\log_2 3})$ [1]。

设 $n = 2l$， $x = x_1 2^l + x_0$， $y = y_1 2^l + y_0$，它们都是 2l 位的整数，则

$$xy = (x_1 2^l + x_0)(y_1 2^l + y_0)$$

$$= x_1 y_1 2^{2l} + [(x_0 + x_1)(y_0 + y_1) - x_1 y_1 - x_0 y_0]2^l + x_0 y_0$$

$x \cdot y$ 通过 3 次 l 位的整数乘法（与普通 2l 位的整数相乘不同）、两次加法和两次减法完成。对于大的 l，加法和减法的时间消耗远小于乘法。这一过程可以

对中间值递归调用，在某一极限条件（比如到达机器所能表示的最大值）时结束，再采用其他传统方法。

对于适当大小的整数，Karatsuba-Ofman 算法是很重要的。为了减少移位次数（用 2^l 和 2^{2l} 乘）和使用更高效的面向字的操作，算法的实际实现可能与传统描述不完全一样。

应该来说这种方法在使用范围上具有一定的局限性，它最为核心的思想是把一个位数较长的数分割成若干个数的级联。分割的深度以及每次分割后数字的位长和最后算法的效率有紧密的联系。由于这种方法并不是本设计所采用的方法，因此在这里就不详细讨论了。

3.3.2 素数域约减

素数域的乘法是由两部分组成的，整数乘法和取模（约减）运算。对于非特殊形式的模 p，求 x mod p 的计算并不比前面介绍的整数乘法运算简单。由于在椭圆曲线密码体制中，素数域乘法的运算量很大，因此很有必要探讨取模运算的高效算法。

Barrett 约减 [71]用低成本的运算代替了高成本的除法来完成取模运算。可以用 Barrett 约减方法直接替换传统的约减方法，然而需要一种高成本的与模相关的计算，因此这种方法适合于对于一个模的多次约减运算。在后面我们还会介绍另外一种素数域乘法的方法，Montgomery 乘法 [72], [73]。这种方法同时完成了乘法和取模的运算，但是它需要对数据进行变换。当多次中间乘法所节省的收益大于输入输出数据变换的消耗时，这种方法是有效的，例如求解模幂的运算。下面我们就分别介绍这两种很有特点的算法。

Barrett 约减

Barrett 约减算法（算法 20） [1], [71]对于给定的正整数 x 和 p 求 x mod p，这种算法是对任意的模 p 都有效的，并不是针对特殊的模 p。用一种低成本的求商[x / p]运算，这种运算采用适当选择的基 b（如 $b = 2^L$，L 根据模来选择，而不是根据 x 来选择）的幂。要计算一个与模相关的量$[b^{2k} / p]$，以使算法适合对同一个模进行多次约减的运算。

算法 20 Barrett 约减

输入：整数 p，$b \geq 3$，$k = [\log_b p] + 1$，$0 \leq x < b^{2k}$，$\mu = [b^{2k} / p]$。

输出：x mod p。

1、$\hat{q} = [[x / b^{k-1}] \cdot \mu / b^{k+1}]$。

2、$r = (x \bmod b^{k+1}) - (\hat{q} \cdot p \bmod b^{k+1})$。

3、若 $r < 0$，则 $r = r + b^{k+1}$。

4、当 $r \geq p$ 时，重复执行 $r = r - p$。

5、返回（r）。

算法 20 的表达形式并不好理解，但它是经过严格的证明得到的结论，下面我们就简要的给出这个算法的证明 [1]。

令 $q = [x / p]$，则 $r = x \bmod p = x - p \cdot q$。算法 20 的步骤 1 计算的是 q 的一个估计值 $\hat{q}$，因为

$$\frac{x}{p} = \frac{x}{b^{k-1}} \cdot \frac{b^{2k}}{p} \cdot \frac{1}{b^{k+1}}$$

注意

$$0 \leq \hat{q} = [\frac{[\frac{x}{b^{k-1}}] \cdot \mu}{b^{k+1}}] \leq [\frac{x}{p}] = q$$

则 $0 \leq \alpha, \beta < 1$，且

$$q = [\frac{([\frac{x}{b^{k-1}}] + \alpha)([\frac{b^{2k}}{p}] + \beta)}{b^{k+1}}]$$

$$\leq [\frac{[\frac{x}{b^{k-1}}] \cdot \mu}{b^{k+1}} + \frac{[\frac{x}{b^{k-1}}] + [\frac{b^{2k}}{p}] + 1}{b^{k+1}}]$$

因为 $x < b^{2k}$ 且 $p \geq b^{k-1}$，由此得到

$$[\frac{x}{b^{k-1}}]+[\frac{b^{2k}}{p}]+1\le(b^{k+1}-1)+b^{k+1}+1=2b^{k+1}$$

和

$$q\le[\frac{[\frac{x}{b^{k-1}}]\cdot\mu}{b^{k+1}}+2]=\hat{q}+2$$

在算法 20 的步骤 2 中计算的 r 必须满足$r\equiv x-\hat{q}\cdot p(\mathrm{mod}\,b^{k+1})$，而$|r|<b^{k+1}$。因此在步骤 3 后，$0\le r<b^{k+1}$且$r=(x-\hat{q}\cdot p)\bmod b^{k+1}$。现因$0\le x-q\cdot p<p$，我们得到

$$0\le x-\hat{q}\cdot p\le x-(q-2)p<3p$$

因为$b\ge 3$且$p<b^k$，我们有$3p<b^{k+1}$。于是$0\le x-\hat{q}\cdot p<b^{k+1}$，并且在步骤 3 后有$r=x-\hat{q}\cdot p$。所以，为了获得$0\le r<p$而有$r=x\bmod p$，在步骤 4 中最多需要两次减法。

给出了算法 20 的简单证明以后，再从计算角度分析这个算法 [1]：

（1）对于基 b 来说，自然的选择就是 $b = 2^L$，其中 L 等于运算系统的字长 W。

（2）与 μ 的计算不同（根据模一次完成），除法运算需要以 b 为基的简单移位。

（3）令$x'=[x/b^{k-1}]$。注意，x'和 μ 最多有$k+1$位基为 b 的数字。在算法的步骤 1 计算$\hat{q}$时丢掉积$x'\cdot\mu$的$k+1$个最低位。给出基为 b 的表达式$x'=\sum x_i'b^i$和$\mu=\sum\mu_j b^j$，于是

$$x'\cdot\mu=\sum_{l=0}^{2k}\underbrace{(\sum_{i+j=l}x_i'\mu_j)}_{w_l}b^l$$

其中 w_l 可能超过 $b-1$。若$b\ge k-1$，则$\sum_{l=0}^{k-1}w_l b^l<b^{k+1}$，因此

$$0\le\frac{x'\mu}{b^{k+1}}-\sum_{l=k-1}^{2k}\frac{w_l b^l}{b^{k+1}}=\sum_{l=0}^{k-2}\frac{w_l b^l}{b^{k+1}}<1$$

由此得出$[\sum_{l=k-1}^{2k}w_l b^l/b^{k+1}]$与$\hat{q}$最多差 1，如果$b\ge k-1$。最多需要$(k^2+5k+2)/2$

次值小于 b 的乘法即可找出这种对 $\hat{q}$ 的评估。

（4）在步骤 2 仅需要 $k+1$ 个 $\hat{q}\cdot p$ 的低有效位。因为 $p<b^k$，所以通过 $(k^2+5k+2)/2$ 次值小于 b 的乘法就可获得这 $k+1$ 个数字。

从算法 20 的证明过程可以看出步骤 1 对 q 的估计值 $\hat{q}$ 的计算最为复杂和关键，而估计值 $\hat{q}$ 的计算是与 b 的选取紧密联系在一起的。步骤 2 将最初的 x mod p 转换成了 $r=(x-\hat{q}\cdot p)\bmod b^{k+1}$ 的计算，由于后面的 $(x-\hat{q}\cdot p)$ 比 x 要小得多，因此计算后者的取模比前者要简单得多。当需要完成大量的取模运算，而它们的模都为 p 时，算法 20 的步骤 1 可以只计算一次，使得计算效率大大提高。如果只完成一次取模运算，步骤 1 的额外开销不一定总能够保证 Barrett 约减比普通的约减效率更高。

Montgomery 乘法

与 Barrett 约减类似，Montgomery 方法 [72], [73], [74], [75], [76], [77]的思想是在普通的约减算法中用简单的运算代替除法运算。这一方法对单个模乘效率并不高，但是用它计算模幂却非常有效，因为计算模幂时要完成给定输入的多次模乘。由于 Montgomery 方法和 Barrett 约减的使用条件同样苛刻，因此在这里仅仅给出 Montgomery 乘法的一般介绍。

在 Montgomery 乘法的计算过程中通常要选用 Montgomery 域上的模数 R，令 R > p 且 gcd (R, p) = 1。Montgomery 乘法运算是对任意一个输入 $\tilde{x}$，$\tilde{y}<pR$，产生 $\tilde{x}\tilde{y}R^{-1}\bmod p$。这里的输入数据 $\tilde{x}$ 和 $\tilde{y}$ 分别是素域的元素 x 和 y 在 Montgomery 域上的表示。通常情况下素域上的模 p 是奇数，因此令 $R=2^{Wt}$，这样使得把 Montgomery 域上的元素还原成素域元素的计算相对简单。

在做 Montgomery 乘法之前，要把素域上的元素 x、y 表示成 Montgomery 域上的元素再进行乘法计算。具体的转换公式为：

x，y∈[0, p)，则

$$\tilde{x}=xR\bmod p$$

$$\tilde{y} = yR \bmod p$$

若令 z = xy，表示 x，y 的乘积，则 $\tilde{x}$ 和 $\tilde{y}$ 的 Montgomery 乘积为：

$$Mont(\tilde{x},\tilde{y}) = \tilde{x}\tilde{y}R^{-1} \bmod p = xyR \bmod p = zR \bmod p = \tilde{z}$$

上面的式子告诉我们 $Mont(\tilde{x},\tilde{y})$ 的结果就是 $\tilde{z}$，它可以继续在 Montgomery 域上进行乘法计算，省去了素域元素和 Montgomery 域元素之间的转换。这个结果告诉我们如果只进行一次乘法运算，采用复杂的变换

$$x \to \tilde{x} = xR \bmod p$$

$$\tilde{x} \to x = \tilde{x}R^{-1} \bmod p$$

是不合算的。然而，对于需要进行大量的模乘计算时只进行一次数据变换是完全可以接受的，如计算模幂。

下面给出具体计算 Montgomery 乘法 $Mont(\tilde{x},\tilde{y})$ 的算法 [1], [72], [73], [74], [75], [76], [77]：

算法 21 Montgomery 乘法算法

输入：素域元素 x、y 的 Montgomery 域表示 $\tilde{x}$ 和 $\tilde{y}$，模 p，$R = 2^{Wt}$。

输出：Montgomery 域的计算结果 $\tilde{z}$。

1、计算 $t = \tilde{x} \cdot \tilde{y}$。

2、计算 $p' = -p^{-1} \bmod R$。

3、计算 $\tilde{z} = (t + (tp' \bmod R)p)/R$。

4、若 $\tilde{z} \geq$ p，则 $\tilde{z} = \tilde{z} -$ p。

5、返回（$\tilde{z}$）。

算法 21 最为精彩的一步出现在步骤 3，将对素数域的模数 p 取模替换成对 R 取模。因为 $R = 2^{Wt}$，为 2 的幂，在二进制数表示中只是最高位为“1”，其余各位全部为“0”的数，这使得对 R 取模的计算以及被 R 除得计算非常简单。进行

一下粗略比较，Montgomery 乘法共用了 $t(t+1)$ 次乘法运算，并且没有用到除，这比 Barrett 约减算法（对于 $b=2^w$ ）的 $t(t+4)+1$ 次乘法还要少。因此可以看出利用 Montgomery 乘法算法计算模幂是十分有效的。

正是由于 Montgomery 算法计算模幂如此高效，因此它还可以用来加速模逆计算，因为有些模逆算法中会出现大量反复的乘法计算。具体的应用在模逆运算这一部分再做详细介绍。

NIST 素数快速约简

除了上面介绍的两种通用的约简算法以外，对于一些特殊的素数域，有更为快速的算法。美国 FIPS 186-2 [78]标准推荐采用 5 个素数上的椭圆曲线进行加密，这 5 个素数的模分别是：

$$p_{192}=2^{192}-2^{64}-1$$

$$p_{224}=2^{224}-2^{96}+1$$

$$p_{256}=2^{256}-2^{224}+2^{192}+2^{96}-1$$

$$p_{384}=2^{384}-2^{128}-2^{96}+2^{32}-1$$

$$p_{521}=2^{521}-1$$

这些素数具有这样的特性：它们都可以表示成少量的 2 的幂的和或差。进而，除了 p_{521} 外，这些素数的上述表达式中的 2 的幂都是 32 的倍数。这一特性使得在字长为 32 位的机器上约简算法的计算特别快。

例如：考虑 $p=p_{192}=2^{192}-2^{64}-1$。令 c 是一个 $0\le c<p^2$ 的整数。则 c 一定可以表示成：

$$c=c_5 2^{320}+c_4 2^{256}+c_3 2^{192}+c_2 2^{128}+c_1 2^{64}+c_0$$

上式可以看成是 c 的 2^{64} 进制的表示式，其中 $c_i\in[0,2^{64}-1]$。我们可以利用以下同余式约简这个式子中的 2 的高次幂项：

$$2^{192}\equiv 2^{64}+1(\mathrm{mod}\,p)$$

$$2^{256} \equiv 2^{128} + 2^{64} (\mathrm{mod}\, p)$$

$$2^{320} \equiv 2^{128} + 2^{64} + 1 (\mathrm{mod}\, p)$$

于是我们得到

$$c \equiv (c_5 2^{128} + c_5 2^{64} + c_5) + (c_4 2^{128} + c_4 2^{64}) + (c_3 2^{64} + c_3) + (c_2 2^{128} + c_1 2^{64} + c_0) (\mathrm{mod}\, p)$$

因此，通过 4 次 192 位的整数 $c_5 2^{128} + c_5 2^{64} + c_5$， $c_4 2^{128} + c_4 2^{64}$， $c_3 2^{64} + c_3$ 和 $c_2 2^{128} + c_1 2^{64} + c_0$ 相加，并重复地减 p（最多减 3 次），直到结果小于 p 为止，便可得到 c mod p。算法 22 给出了详细的过程 [1]。

算法 22 模 $p_{192} = 2^{192} - 2^{64} - 1$ 的快速约减

输入：$c = (c_5, c_4, c_3, c_2, c_1, c_0)$ 为 2^{64} 进制整数，$0 \le c < {p_{192}}^2$。

输出：$c \bmod p_{192}$。

1、定义 192 位整数：

$s_1 = (c_2, c_1, c_0)$， $s_2 = (0, c_3, c_3)$，

$s_3 = (c_4, c_{41}, 0)$， $s_4 = (c_5, c_5, c_5)$。

2、返回（$s_1 + s_2 + s_3 + s_4 \bmod p_{192}$）。

除了 p_{192} 以外，p_{224}，p_{256}，p_{384} 和 p_{521} 这几个特殊的素数域的约减也都有类似的快速方法，在本节就不一一给出了。

3.3.3 二进制域乘法

由于在二进制域 F_{2^m} 中，所有元素全部表示成次数最高为 $m-1$ 次的多项式，因此乘法运算也有不同的形式。下面我们介绍几种用于二进制域上的乘法算法。

移位加算法

一种比较简单的方法是用移位加的方法来实现二进制域上的乘法 [1]，这种

方法建立在下面这个表达式的基础之上：

$$a(x)\cdot b(x)=a_{m-1}x^{m-1}b(x)+\cdots+a_2x^2b(x)+a_1xb(x)+a_0b(x)$$

在算法 23 中对 i 实施迭代，计算 $x^ib(x)\bmod f(x)$，并且若 $a_i=1$，则把结果加到累加器 c 上。

若 $b(x)=b_{m-1}x^{m-1}+\cdots+b_2x^2+b_1x+b_0$，则

$$\begin{aligned}b(x)\cdot x&=b_{m-1}x^m+b_{m-2}x^{m-1}+\cdots+b_2x^3+b_1x^2+b_0x\\&=b_{m-1}r(x)+(b_{m-2}x^{m-1}+\cdots+b_2x^3+b_1x^2+b_0x)\qquad(\bmod f(x))\end{aligned}$$

这样，$b(x)\cdot x\bmod f(x)$ 就可以通过对 b(x)的向量表示进行左移来实现，边移位边判断，若最高位 b_{m-1} 为 1，则把 r(x)加到 b(x)上。其中 r(x)为 x^m 对 f(x)取模以后的结果：

$$r(x)=x^m\bmod f(x)=f_{m-1}x^{m-1}+\cdots+f_2x^2+f_1x+f_0$$

算法 23　从右向左移位加计算二进制域 F_{2^m} 上的乘法

输入：次数低于 m 的二进制多项式 $a(x)$，$b(x)$。

输出：$c(x)=a(x)\cdot b(x)\bmod f(x)$。

1、$c(x)=0$。

2、对于 i 从 0 到 $m-1$，重复执行

2.1　$b(x)=b(x)\cdot x\bmod f(x)$。

2.2　若 $a_i=1$，则 $c(x)=c(x)+b(x)$。

3、返回（$c(x)$）。

尽管算法 23 在软件实现领域应用得非常广泛，但是这个算法同时也非常适合于硬件实现，并且效率很高。下面我们分析一下算法 23 适合于硬件实现的原因：

第一，同时完成了多项式乘法和约减（取模）两个运算，这一点主要体现在

算法中的步骤 2.1；

第二，算法中只含有移位和按位异或两种运算，而这两种运算用硬件来实现都很容易；

第三，在运算中所有操作数的长度都为 m 位的二进制数，用硬件实现起来结构整齐。

Comb 多项式乘法

算法 23 实现起来非常简单，但是很明显要经过 m 次的循环才能计算出最后的乘积，如果 m 的值很大，运算速度较慢。Comb 多项式乘法 [79], [80]是在其基础上发展起来的一种算法，可有效的提高运算速度。

如果把二进制域上的任意元素 a 的多项式表示

$$a(x) = a_{m-1}x^{m-1} + \cdots + a_2x^2 + a_1x + a_0$$

与一个 m 维向量 $a = (a_{m-1}, \cdots, a_2, a_1, a_0)$ 线对应。像素域元素的二进制表示一样，可以令 $t = [m/W] + 1$，$s = Wt - m$。可以用一个由 t 个 W 字的数组 $A = (A[t-1], \cdots A[2], A[1], A[0])$ 来存储元素 a，最低有效位 a_0 存储到 A[0]中，并且 A[t – 1]的最左 s 位没有用（总是设置为“0”）。具体的表示可以参看表 3.2。

表 3.2　用 W 位的字表示二进制域元素 a

	A[t – 1]		A[1]	A[0]
	$a_{m-1}\cdots a_{(t-1)W}$	…	$a_{2W-1}\cdots a_{W+1}a_W$	$a_{W-1}\cdots a_1a_0$

从右向左的 comb 多项式乘法（算法 23）建立在下面的观察基础之上：若对于某些 k，k∈[0，W – 1]，$b(x)\cdot x^k$ 已经计算出来，则通过在 $b(x)\cdot x^k$ 的表示向量的右边附加 j 个“0”字，便很容易地获得 $b(x)\cdot x^{Wj+k}$。算法 24 从右向左处理 A 字的各位，如表 3.3 所示。

表 3.3　A 字的二维数组表示（从右向左）。

表中的参数为 W = 32，m = 163

A[0]	a_{31}	…	a_2	a_1	a_0
A[1]	a_{63}	…	a_{34}	a_{33}	a_{32}
A[2]	a_{95}	…	a_{66}	a_{65}	a_{64}
A[3]	a_{127}	…	a_{98}	a_{97}	a_{96}
A[4]	a_{159}	…	a_{130}	a_{129}	a_{128}
A[5]			a_{162}	a_{161}	a_{160}

表中的参数 $m=163$，$W=32$。我们采用下列符号：若 $C=(C[n],\cdots,C[2],C[1],C[0])$ 是一个数组，则 C{j} 表示截短数组 $(C[n],\cdots,C[j+1],C[j])$。

算法 24　从右向左的 comb 多项式乘法

输入：次数低于 m 的二进制多项式 $a(x)$，$b(x)$。

输出：$c(x)=a(x)\cdot b(x)$。

1、C = 0。

2、对于 k 从 0 到 W – 1，重复执行

2.1 对于 j 从 0 到 t – 1，重复执行

若 A[j]的第 k 位是 1，则 B + C{j}。

2.2 若 $k\neq(W-1)$，则 B = B · x。

3、返回（C）。

从右向左的 comb 多项式乘法按从左向右的方向处理 a 的位，如下所示：

$$a(x)\cdot b(x)=(\cdots((a_{m-1}b(x)x+a_{m-2}b(x))x+a_{m-3}b(x))x+\cdots+a_1b(x))x+a_0b(x)$$

算法 25 是对这一方法的改进，其中字 A 的处理按从左向右的方向进行。表 3.4 显示了这种改进，表中的参数 m = 163，W = 32。

表 3.4　A 字的二维数组表示（从左向右）。

表中的参数为 W = 32，m = 163

a_{31}	…	a_2	a_1	a_0	A[0]
a_{63}	…	a_{34}	a_{33}	a_{32}	A[1]
a_{95}	…	a_{66}	a_{65}	a_{64}	A[2]
a_{127}	…	a_{98}	a_{97}	a_{96}	A[3]
a_{159}	…	a_{130}	a_{129}	a_{128}	A[4]
		a_{162}	a_{161}	a_{160}	A[5]

算法 25　从左向右的 comb 多项式乘法

输入：次数低于 m 的二进制多项式 $a(x)$，$b(x)$。

输出：$c(x) = a(x) \cdot b(x)$。

1、C = 0。

2、对于 k 从 W – 1 到 0，重复执行

2.1 对于 j 从 0 到 t – 1，重复执行

若 A[j]的第 k 位是 1，则 B + C{j}。

2.2 若 $k \neq 0$，则 C = C · x。

3、返回（C）。

算法 24 和算法 25 都比算法 23 的计算速度快，因为它们只有少量的向量移位（乘以 x）。算法 24 比算法 25 还要快，因为算法 24 的向量移位涉及 t 个字的数组 B（它可以增长到 t + 1 个字），而算法 25 的向量移位涉及 2t 个字的数组 C。

Comb 算法是由 López 和 Dahab [80]提出来的，并且是以 Lim 和 Lee [79]提出的可修改用于二进制域的指数方法为基础的。在算法 24 和算法 25 中的步骤 2.1 是一个循环运算，在实现的过程中对于 j 从 0 到 t – 1 的所有 A[j]是可以同时进行计算的。这也是这两个算法速度快的根本原因，当然代价是增加了控制的复杂性。不过算法 24 和算法 25 在应用上的最大困难是它们的计算结果还需要进行对既约

多项式 f(x)的取模运算。在素数域中提到的 Karatsuba-Ofman 乘法算法 [1]也可以应用到二进制域中来，同样面临取模的运算。二进制域的取模运算和素域取模运算相比有哪些不同呢？下面我们就详细介绍一下二进制域的取模运算。

3.3.4 二进制域约减

二进制域约减的对象是多项式 c(x)，它是 a(x)和 b(x)的乘积，因此是一个次数最高为 2m – 2 次的多项式。而在二进制域中的模为既约多项式 f(x)。

任意约减多项式

先来回忆既约多项式 $f(x)=x^m+r(x)$，其中 r(x)是一个次数最高为 $m-1$ 次的二进制多项式。算法 26 [1]对 c(x)模 f(x)约减，一次处理一位，从最低位开始。可以这样处理的理由如下：

$$\begin{aligned} c(x) &= c_{2m-2}x^{2m-2}+\cdots+c_m x^m+c_{m-1}x^{m-1}+\cdots+c_1x+c_0 \\ &= (c_{2m-2}x^{m-2}+\cdots+c_m)r(x)+c_{m-1}x^{m-1}+\cdots+c_1x+c_0 \qquad (\mathrm{mod}\, f(x)) \end{aligned}$$

通过预计算多项式 $x^k r(x)$，$0\le k\le W-1$，约减运算被加速。若多项式 r(x)的次数很低，或 f(x)是一个三项式，则所需要的存储空间很小，而且在步骤 2.1 中涉及 $x^k r(x)$ 的加法的速度就很快，不过这只是一种特殊情况。用到下列符号：若 $C=(C[n],\cdots,C[2],C[1],C[0])$ 是一个数组，则 $C\{j\}$ 表示截短数组 $(C[n],\cdots,C[j+1],C[j])$。

算法 26 模约减（一次一位）

输入：次数最高位 $2m-2$ 的二进制多项式 $c(x)$。

输出：$c(x) \bmod f(x)$。

1、预计算。计算 $u_k(x)=x^k r(x)$，$0\le k\le W-1$。

2、对于 i 从 $2m-2$ 到 m，重复执行

2.1 若 $c_i = 1$，则

令 $j = [(i-m)/W], k = (i-m) - Wj$。

$u_k(x) + C\{j\}$。

3、返回（$C[t-1], \cdots, C[2], C[1], C[0]$）。

NIST 约减方法

若 f(x)是一个三项式，或是一个中间项互相靠近的五项式，则约减运算 c(x) mod f(x)可以一次处理一个字。

例如：设 m = 163，W = 32（因此 t = 6），考虑 c(x)模 $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ 的字 C[9]的约减。字 C[9]表示多项式为 $c_{319}x^{319} + \cdots + c_{289}x^{289} + c_{288}x^{288}$。我们有

$$x^{288} = x^{132} + x^{131} + x^{128} + x^{125} (\mathrm{mod}\, f(x))$$

$$x^{289} = x^{133} + x^{132} + x^{129} + x^{126} (\mathrm{mod}\, f(x))$$

$$\vdots$$

$$x^{319} = x^{163} + x^{162} + x^{159} + x^{156} (\mathrm{mod}\, f(x))$$

考虑上述同余式右边的四列（竖着看），我们发现 C[9]的约减可以通过把 C[9]四次加到 C 上来实现。而 C[9]的最低位四次分别加到 C 的 132，131，128 和 125 位上。这一过程如图 3.1 所示。

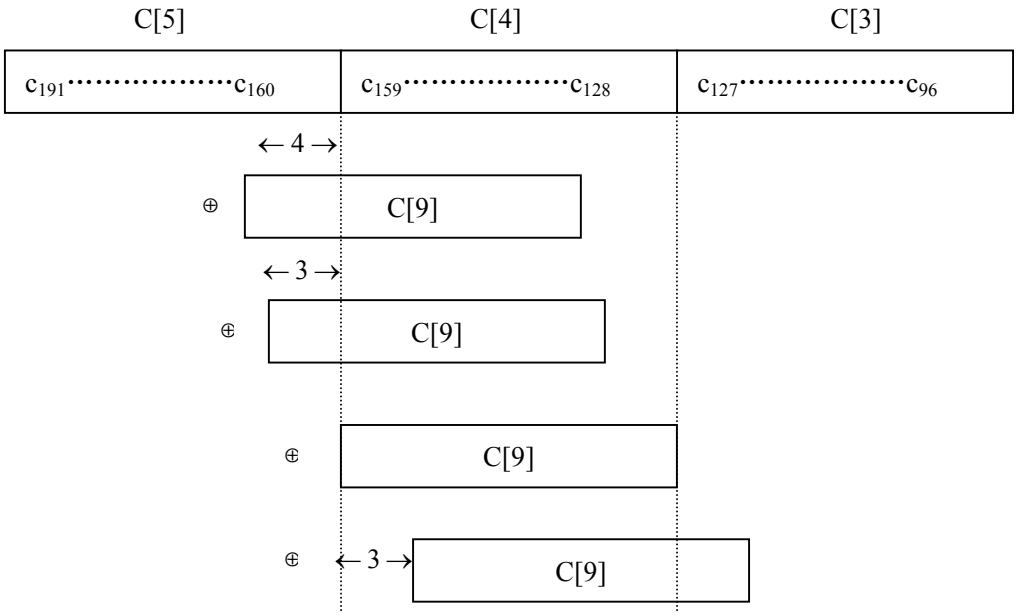

图 3.1　约减 32 位字 C[9]模 $f(x)=x^{163}+x^7+x^6+x^3+1$

下面我们给出一种快速模约减的方法，下列约减多项式是 NIST 在 FIPS 186-2 [78]中推荐的：

$$f(x)=x^{163}+x^7+x^6+x^3+1$$

$$f(x)=x^{233}+x^{74}+1$$

$$f(x)=x^{283}+x^{12}+x^7+x^5+1$$

$$f(x)=x^{409}+x^{87}+1$$

$$f(x)=x^{571}+x^{10}+x^5+x^2+1$$

这 5 个多项式的约减都可以采用与图 3.1 类似的方法，在这里只给出前两个多项式约减的具体算法。它们一个是五项式，一个是三项式。因为在二进制域的椭圆曲线加密算法中，既约多项式 f(x)为三项式或五项式的情况是比较常见的，因此我们详细讨论这种多项式的快速约减方法。只要是这两类多项式都可以仿照这种方法，它们都要比算法 26 快得多，并且不需要很多存储空间。在算法中假设字宽为 W = 32。

算法 27 模 $f(x)=x^{163}+x^7+x^6+x^3+1$ 的快速约减（W = 32）

输入：次数最高位 324 的二进制多项式 c(x)。

输出：c(x) mod f(x)。

1、对于 i 从 10 到 6，重复执行　　{约减 $C[i]x^{32i} \bmod f(x)$}

1.1 T = C[i]。

1.2 C[i – 6] = C[i – 6] ⊕ (T << 29)。

1.3 C[i – 5] = C[i – 5] ⊕ (T << 4) ⊕ (T << 3) ⊕ T ⊕ (T >> 3)。

1.4 C[i – 4] = C[i – 4] ⊕ (T >> 28) ⊕ (T >> 29)。

2、T = C[5] >> 3。　　{提取 C[5]的 3~31 位}

3、C[0] = C[0] ⊕ (T << 7) ⊕ (T << 6) ⊕ (T << 3) ⊕ T。

4、C[1] = C[1] ⊕ (T >> 25) ⊕ (T >> 26)。

5、C[5] = C[5] & 0x7。　　{清除 C[5]的要约减的位}

6、返回（C[5], C[4], C[3], C[2], C[1], C[0]）。

算法 28 模 $f(x)=x^{233}+x^{74}+1$ 的快速约减（W = 32）

输入：次数最高位 464 的二进制多项式 c(x)。

输出：c(x) mod f(x)。

1、对于 i 从 15 到 8，重复执行　　{约减 $C[i]x^{32i} \bmod f(x)$}

1.1 T = C[i]。

1.2 C[i – 8] = C[i – 8] ⊕ (T << 23)。

1.3 C[i – 7] = C[i – 7] ⊕ (T >> 9)。

1.4 C[i – 5] = C[i – 5] ⊕ (T << 1)。

1.5 C[i – 4] = C[i – 4] ⊕ (T >> 31)。

2、T = C[7] >>9。　　{提取 C[7]的 9~31 位}

3、C[0] = C[0] ⊕ T。

4、C[2] = C[2] ⊕ (T << 10)。

5、C[3] = C[3] ⊕ (T >> 22)。

6、C[7] = C[7] & 0x1FF。　　{清除 C[7]的要约减的位}

7、返回（C[7], C[6], C[5], C[4], C[3], C[2], C[1], C[0]）。

算法 27 [1]和算法 28 [1]中的移位运算虽然复杂，但在图 4.1 的帮助下就好理解了。值得注意的是图 4.1 只是针对算法 27，不过可以仿照图 4.1 画出与算法 28 所对应的图，它们除了在移位上不同以外，根本思想是一致的。NIST 约减方法速度的确很快，不过遗憾的是每一个约减多项式对应一个算法，通用性不好，这点在硬件设计上应加以考虑。

3.3.5 素数域和二进制域乘法比较

通过前面的介绍我们了解到无论是计算素数域乘法还二进制域乘法，都要分两步完成：第一，计算有限域上两个元素的乘积；第二，约减。在素域中有 Barrett 约减算法和 Montgomery 算法，在二进制中有 NIST 快速约减。但是这些方法有的运算过程很复杂，占用资源很多，有的只适用于某些特殊情况，考虑到在 ECC 密码算法中几乎每次乘法完成以后都需要进行约减运算，因此一定要选择一种约减效率很高的有限域乘法算法。

算法 23 移位加算法满足设计的要求，这个算法最大的优点就是可以同时完成多项式乘法和约简（取模）两个运算，节省了大量的计算时间和资源。但是，这个算法只适合于二进制域中的多项式乘法，如果将算法 23 做出改进，使得新算法也可以完成素数域上的乘法和约减运算，将会提高整个 ECC 运算的效率。

将二进制域多项式乘法移植到素数域上时面临如下三个难题：

（1） 素数域和二进制域上的元素表示形式不同；

（2） 素数域和二进制域上的两个元素的加法运算不同；

（3） 素数域和二进制域上的元素的约减运算不同。

我们设计的算法 29 很好的解决了上面的三个难题。

算法 29 同时完成乘法和约减运算的素域乘法算法

输入：素域两个元素 a，b（元素 a，b 都是位数小于 m 位的二进制数表示形式）。

输出：$c = a \cdot b \bmod p$ 。

1、c = 0。

2、重复执行

2.1 若 $a_0 = 1$，则 c = c + b。

2.2 若 c ≥ p，则 c = c – p。

2.3 b = b << 1。

2.4 若 b ≥ p，则 b = b – p。

2.5 a = a >> 1。

2.6 若 a = 0，则执行步骤 3。

3、返回（c）。

我们仔细分析一下算法 29，就可以发现它不但很好的解决了上面提到的三个难题，而且还将算法 23 做出了一定的改进，使得算法 29 更适合于硬件实现。

在做素数域的运算时素数域的元素都是用二进制数的形式表示的，这一点使得把算法 23 改进成算法 29 变得很容易。素数域的加法比二进制域的加法要复杂，不过我们可以利用素数域加法运算中已有的资源，因此这并不会增加太多的运算成本。而素数域中的取模运算是最难实现的，在这方面算法 29 做出了如下几个方面的改进：

1、增加了两次取模运算。算法 23 中的取模运算与移位运算合并执行，在二进制域中可以；算法 29 针对素数域取模的运算特点把取模运算单独作为一个步骤，并且对移位以后的 b 和累加器 c 都进行判断取模。做出这样的调整是必要的，因为二进域加法的结果是不会超界的，但是素数域加法的结果一定要进行取模运算。

2、运算次数的改进。当 a 和 b 两个寄存器的位数为 m 位时，算法 23 将固定的循环 m 次。即算法 23 的运算次数受到硬件结构的束缚，与输入数据的位数没有直接关系，造成算法 23 的效率不够高，且运算灵活性差。在算法 29 的步骤 2 中对这一情况做出调整。通过对 a 的移位和判断来控制运算次数（参看算法 29 的步骤 2.5 和 2.6），这使得运算次数更加灵活，在同样的硬件结构下虽然算法 29 最多仍需完成 m 次循环运算，但算法 29 可以随着输入数据的变化而自动调整循环次数。当输入数据 a 不足 m 位时，存储数据 a 的 m 位寄存器前几位将会置“0”，

而算法 29 对于高位的“0”将会忽略，并不做任何运算。这样算法 29 实际运算的次数将会由输入数据的位数决定，而与存储器的位数无关。这样使得算法 29 的运算灵活性非常强。对于 m 位的输入数据，算法将会完成 m 次循环运算，在每次循环运算中，最多将完成两次移位运算、一次加法运算、一次减法运算和一次比较运算。

算法 29 的诸多优点与算法 23 结合起来，这样将会得到效率更高的关于二进制域乘法的计算方法，这就是我们提出的算法 30。算法 30 最多完成 m 次循环运算，每次循环将会完成两次移位运算和两次按位异或运算。相对素数域上的乘法运算，二进制域上的乘法运算速度会更快。

算法 30　二进制域上的乘法算法

输入：二进制域上的两个元素 a，b（元素 a，b 都是位数小于 m 位的二进制数，为元素 a 和 b 的多项式表示）。

输出：$c = a \cdot b \bmod f$ （f 为二进制域上的取模多项式）。

1、c = 0。

2、重复执行

2.1 若 $a_0 = 1$，则 c = c ⊕ b。

2.2 b = （b << 1） ⊕ f。

2.3 a = a >> 1。

2.4 若 a = 0，则执行步骤 3。

3、返回（c）（$c = a \cdot b \bmod f$)。

通过前面的分析，可以看出算法 29 和算法 30 分别可以计算素数域和二进制域上的乘法运算，这两个算法全部是我们自己通过参考已有的算法，结合我们硬件实现的要求，自己设计出的算法。对于这两个算法的创新主要有以下三点：

1、将二进域上的一种高效的乘法算法“移植”到了素数域上，即将算法 23 改进成算法 29。

2、乘法和约减（取模）运算同时完成，既提高运算速度，又节省硬件资源。这一点在前面多次强调过了，这也是我们最为重要的改进。速度上的提高毋庸置

疑，并且我们省去了取模电路的设计，同时也节省了硬件资源。

3、算法计算能力增强，控制条件简化。从而简化了电路控制条件，使得乘法运算电路更加灵活，可移植性更强。

3.4 有限域平方

在这一部分我们专门来介绍有限域中的平方运算，不管是在素数域中还是二进制域中，利用前面介绍的乘法运算算法都可以完成平方运算，但是由于平方运算的两个因数一样，因此有更简洁的算法。

3.4.1 素数域平方运算

素数域 F_p 上的元素 a 的平方可以这样计算，首先把 a 作为整数计算出平方，然后求模 p 的余数，即先计算平方再取模，与素数域乘法运算一致。对算法 19 进行直接修改便得到如下的求整数平方的算法，算法 31 [1]比算法 19 减少了约一半的 W 位字乘（$(UV) = A[i] \cdot B[i]$）运算。具体的算法如下。

算法 31 整数平方

输入：整数 $a \in [0, p - 1]$。

输出：$c = a^2$。

1、$R_0 = 0$，$R_1 = 0$，$R_2 = 0$。

2、对于 k 从 0 到 $2t - 2$，重复执行 3

2.1 对于集合$\{(i, j) \mid i + j = k, 0 \le i \le j \le t - 1\}$中的每一个元素，重复执行

$(UV) = A[i] \cdot A[i]$。

若（$i < j$），则$(\varepsilon, UV) = (UV) \cdot 2$，$R_2 = R_2 + \varepsilon$。

$(\varepsilon, R_0) = R_0 + V$。

$(\varepsilon, R_1) = R_1 + U + \varepsilon$。

$R_2 = R_2 + \varepsilon$。

2.2 $C[k] = R_0$，$R_0 = R_1$，$R_1 = R_2$，$R_2 = 0$。

3、$C[2t - 1] = R_0$。

4、返回（c）。

在步骤 2.1 中，2 与（2W）位的量（UV）相乘，得到的一个（2W + 1）位的值即为(ε, UV)。这个运算可以通过两个 W 位字带进位移位或用两个 W 位字进行带进位加法来实现。这个步骤可以重新改写，以使每个输出字 C[k]最多需要一次用 2 乘运算，但这样需要两个累加器和相应的累加步骤。

在完成平方运算之后还需要进行取模运算，这就和素数域乘法完成以后的取模完全一样，前面已经给出了具体的算法，这里就不再赘述了。

3.4.2 二进制域平方运算

在二进制域中，所有元素都表示为一个多项式，因此平方运算完成的就是一个多项式平方。因为二进制多项式的平方是一种线形操作，所以它比任意两个多项式相乘快得到多。

若 $a(x) = a_{m-1}x^{m-1} + \cdots + a_2x^2 + a_1x + a_0$，则

$$a(x)^2 = a_{m-1}x^{2m-2} + a_{m-2}x^{2m-4} + \cdots + a_2x^4 + a_1x^2 + a_0$$

这是由于二进制多项式的系数为（0，1），即所有系数都是对 2 取模后的余数，因此 $a(x)^2$ 的运算结果每隔一项就会有一个“0”出现。因此 $a(x)^2$ 的二进制表示可以通过以下方式获得：往 a(x)的二进制表示中相邻的位之间插入一个 0，如图 3.2 所示。

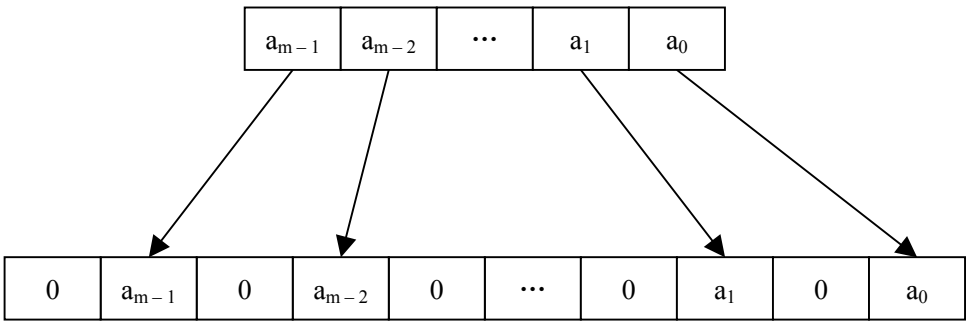

图 3.2　二进制多项式 $a(x) = a_{m-1}x^{m-1} + \cdots + a_2x^2 + a_1x + a_0$ 的平方

正是由于多项式平方算法的特点，有时为了便于处理，可以用一个 512 字节的表 T 进行预计算，把 8 位的多项式变换扩充为 16 位的对应多项式。算法 32 给出了 W = 32 的这种处理过程。这种方法适合于对计算速度要求比较高的设计中。

算法 32 多项式平方（采用预计算的方法，字长 W = 32）

输入：次数低于 m 的二进制多项式 a(x)。

输出：$c(x) = a(x)^2$。

1、预计算。对于每个字节 $d = (d_7, \cdots, d_1, d_0)$，产生 16 位的量 $T(d) = (0, d_7, \cdots, 0, d_1, 0, d_0)$。

2、对于 i 从 0 到 t – 1，重复执行

2.1 令 $A[i] = (u_3, u_2, u_1, u_0)$，其中 u_j 是一个字节。

2.2 $C[2i] = (T(u_1), T(u_0))$，$C[2i + 1] = (T(u_3), T(u_2))$。

3、返回（c(x)）。

计算多项式的平方运算量很小，特别是使用预计算的方法来计算，速度非常快，其代价是需要用到大量的存储空间来存储预计算的结果。不过当完成了多项式平方运算后还要进行取模运算，这一点在多项式乘法运算中已经做出了详细介绍。

3.5 有限域求逆

在有限域的各种运算当中，求逆运算是最为复杂，最耗资源，运算时间最长的一种运算。求逆运算效率的高低将直接影响到椭圆曲线加密算法的运算效率。目前国内外的学者针对求逆运算的研究非常多，本文的作者也发表了几篇关于求逆运算的文章 [10]。在这一部分我们将详细介绍素数域和二进制域上求逆运算的常见算法。并通过对各种算法的比较和改进，提出一种可以同时支持素数域和二进制域的双域求逆算法。

3.5.1 素数域求逆算法

在素数域 F_p 上的非零元素 a 的逆记为 $a^{-1} \bmod p$，或直接简记为 a^{-1}。即在素数域 F_p 上存在惟一的一个元素 x，使得 $a \cdot x \equiv 1 (\bmod p)$，则元素 x 为 a 的逆 a^{-1}。常见的计算 a^{-1} 的算法有以下两种，我们简要的介绍一下：

1、扩展的整数 Euclidean 算法 [1], [81]。

2、Montgomery 求逆算法 [82], [83], [84], [85], [86], [87], [88]。

扩展的整数 Euclidean 算法

令 a 和 b 为非零整数，则 a 和 b 的最大公因子表示为 $\gcd(a,b)$。a 和 b 的最大公因子 $\gcd(a,b)$ 是能同时整除 a 和 b 的最大整数 d。计算 $\gcd(a,b)$ 的算法可根据下面的结论得出：

如果 a 和 b 是正整数，则对于所有的整数 c，都有

$$\gcd(a,b) = \gcd(b - ca, a)$$

在计算正整数 a 和 b（$b \geq a$）的最大公因子 $\gcd(a,b)$ 的传统 Euclidean 算法中，用 a 除 b，得到满足 $b = qa + r$，$0 \leq r < a$ 的商 q 和余数 r。因此有 $\gcd(a,b) = \gcd(r,a)$。这样确定 $\gcd(a,b)$ 的问题，就简化为确定 $\gcd(r,a)$ 的问题，其中变量(r，a)小于原来的变量(a，b)。这一过程一直持续进行，直到变量中一个为 0 时结束，因为 $\gcd(0,d) = d$，所以得到最终结果 $\gcd(a,b)$。这一算法必定终止，因为非负余数序列是严格递减的，这个算法是有效的。设 a 的二进制长度为 m，则算法中除法的次数最多为 2m 次。下面给出传统 Euclidean 算法的具体表示形式 [1]。

算法 33 传统的 Euclidean 算法

输入：正整数 a 和 b，且 $a \le b$。

输出：$d = \gcd(a,b)$。

1、$u = a$，$v = b$。

2、当 $r \ne 0$ 时，重复执行

2.1 $q = [v/u]$。

2.2 $r = v - qu$。

2.3 $v = u$，$u = r$。

3、$d = v$。

4、返回（d）。

算法 33 的计算过程非常简单，但是在求 $d = \gcd(a,b)$ 的过程中可以找到求 a^{-1} 的方法。在算法 33 的计算过程中得到一系列的等式：

$$b = q_1 a + r_1$$

$$a = q_2 r_1 + r_2$$

$$r_1 = q_3 r_2 + r_3$$

$$\vdots$$

$$r_{i-2} = q_i r_{i-1} + r_i$$

这些等式都是计算 $d = \gcd(a,b)$ 的过程中得到的，其中 q_i 和 r_i 分别是步骤 2.1 和步骤 2.2 在每次计算后得到的结果。

最后一个通用的等式 $r_{i-2} = q_i r_{i-1} + r_i$ 也可以改写成 $r_i = r_{i-2} - q_i r_{i-1}$。如果把步骤 2.2 计算的余数 r_i 用一个通式来表示：

$$r_i = ax_i + by_i$$

则可以得到下面的结论：

$$r_i = r_{i-2} - q_i r_{i-1}$$

$$= (ax_{i-2} + by_{i-2}) - q_i(ax_{i-1} + by_{i-1})$$

$$= a(x_{i-2} - q_i x_{i-1}) + b(y_{i-2} - q_i y_{i-1})$$

$$= ax_i + by_i$$

从上面的式子中可以找到 x_i 和 y_i 的递推公式：

$$x_i = x_{i-2} - q_i x_{i-1}$$

$$y_i = y_{i-2} - q_i y_{i-1}$$

它们和 r_i 的递推公式一样。因此完全可以用计算 r 的算法来求出 x_i 和 y_i 的取值。

通过上面的推导，我们知道 Euclidean 算法还可以扩展用于寻找出两个整数 x 和 y，使得

$$ax + by = d, (d = \gcd(a,b))$$

算法 33 给出了扩展的整数 Euclidean 算法的运算过程，在这个算法中保持下列关系式不变：

$$ax_1 + by_1 = u, ax_2 + by_2 = v, u \le v$$

当 $u = 0$ 时算法终止，此时 $v = \gcd(a,b)$，并且 $x = x_2$， $y = y_2$，满足 $ax + by = d$。

算法 34 扩展的整数 Euclidean 算法

输入：正整数 a 和 b，且 $a \le b$。

输出：$d = \gcd(a,b)$，满足 $ax + by = d$ 的整数 x 和 y。

1、$u = a$， $v = b$。

2、$x_1 = 1$， $y_1 = 0$， $x_2 = 0$， $y_2 = 1$。

3、当 $u \neq 0$ 时，重复执行

3.1 $q = [v/u]$， $r = v - qu$， $x = x_2 - qx_1$， $y = y_2 - qy_1$。

3.2 $v = u$， $u = r$， $x_2 = x_1$， $x_1 = x$， $y_2 = y_1$， $y_1 = y$。

4、$d=v$， $x=x_2$， $y=y_2$。

5、返回(d,x,y)。

设 p 是一个素数，且$a\in[1,p-1]$， $\gcd(a,p)=1$。若上面的算法 34 [1]对于输入(a,p)执行，则在步骤 3.1 遇到的最后的非零余数$r=1$。在步骤 3.2 中 u，x_1，x_2被更新，以满足$ax_1+py_1=u$，而$u=1$。因此$ax_1\equiv 1 \bmod p$，而$a^{-1}=x_1 \bmod p$。注意，确定 x_1并不需要 y_1和 y_2。根据这些分析，就可以导出素域 F_p上求逆运算的算法 35 [1]。

算法 35 利用扩展的 Euclidean 算法求 F_p上的逆

输入：素数 p 和 a，其中$a\in[1,p-1]$。

输出：$a^{-1} \bmod p$。

1、$u=a$， $v=p$。

2、$x_1=1$， $x_2=0$。

3、当$u\neq 0$时，重复执行

3.1 $q=[v/u]$， $r=v-qu$， $x=x_2-qx_1$。

3.2 $v=u$， $u=r$， $x_2=x_1$， $x_1=x$。

4、返回（$x_1 \bmod p$）。

算法 35 的一个缺点是在步骤 3.1 需要复杂的除法运算。所以为了使算法 35 运算效率更高，同时也更加适合于硬件实现，通常把除法运算改成简单的移位操作（乘 2 或除 2）和减法运算。素数域上的元素 a 和模数 p 都是用二进制数表示的，所以移位操作实现起来非常简单。向右移位（除 2）这种方式使得操作数不断减小，运算简单，下面我们就介绍这种方法。而向左移位（乘 2）则在二进制域的求逆运算中用的较多。

算法 36 [1]是改进后的求 gcd 的算法，在该算法中对操作数进行除 2，即向右移位的运算。在步骤 3.1 的每次迭代之前，u 和 v 中最多有一个是奇数，因此在步骤 3.1 和步骤 3.2 的除 2 操作不改变 $\gcd(u,v)$ 的值。在每次迭代中步骤 3.1 和步骤 3.2 执行之后，u 和 v 都变成奇数。并且在步骤 3.3 之后 u 和 v 中恰好有一个成为偶数。因此，步骤 3 的每一次迭代都缩减了 u 或 v 的二进制长度。因此最多迭代 2m 次，其中 m 是 a 或 b 的二进制长度的最大值。

算法 36 改进后的 gcd 算法

输入：正整数 a 和 b。

输出：$\gcd(a,b)$。

1、$u=a$，$v=b$，$e=1$。

2、当 u 和 v 都是偶数时，重复执行

$u=u/2$，$v=v/2$，$e=2e$。

3、当 $u\neq 0$ 时，重复执行

3.1 当 u 是偶数时，重复执行 $u=u/2$。

3.2 当 v 是偶数时，重复执行 $v=v/2$。

3.3 若 $u\geq v$，则 $u=u-v$；否则，$v=v-u$。

4、返回（$e\cdot v$）。

算法 37 [1]通过寻找满足 $ax+py=1$ 的整数 x 来计算 $a^{-1} \bmod p$。算法保持下列关系式不变：

$$ax_1+py_1=u, ax_2+py_2=v$$

其中 y_1 和 y_2 不被计算。当 $u=1$ 或 $v=1$ 时算法终止。对于前一种情况，$ax_1+py_1=1$，因此 $a^{-1}=x_1 \bmod p$。对于后一种情况，$ax_2+py_2=1$，因此 $a^{-1}=x_2 \bmod p$。

算法 37 改进后的 F_p 上的求逆算法

输入：素数 p 和 a，其中 $a\in[1,p-1]$。

输出：$a^{-1} \bmod p$。

1、$u = a$，$v = p$。

2、$x_1 = 1$，$x_2 = 0$。

3、当$u \neq 1$和$v \neq 1$时，重复执行

3.1 当 u 是偶数时，重复执行

$u = u/2$，

若 x_1 是偶数，则 $x_1 = x_1/2$；否则，$x_1 = (x_1 + p)/2$。

3.2 当 v 是偶数时，重复执行

$v = v/2$，

若 x_2 是偶数，则 $x_2 = x_2/2$；否则，$x_2 = (x_2 + p)/2$。

3.3 若$u \geq v$，则$u = u - v$，$x_1 = x_1 - x_2$；

否则，$v = v - u$，$x_2 = x_2 - x_1$。

4、若$u = 1$，则返回（$x_1 \bmod p$）；否则返回（$x_2 \bmod p$）。

把求 F_p 上元素逆的改进算法的初始条件$x_1 = 1$修改为$x_1 = b$，便可得到计算$b/a = ba^{-1} \bmod p$的除法算法。修改后的算法和原来算法在执行时间上是相同的，因为在求逆算法中少量的迭代后 x_1 的二进制长度就是全长了。算法 37 的计算效率是非常高的，该算法的计算时间最多为相同位数的乘法算法的两倍。

Montgomery 求逆算法

在前面的章节介绍素数域的约简算法时已经介绍了 Montgomery 算法。这种方法的基本思想是对于特别选择的 R，用低成本的$xR^{-1} \bmod p$运算代替了复杂的$x \bmod p$运算。Montgomery 算法可以看作是对表示式$\tilde{x} = xR \bmod p$的运算，我们知道这个算法特别适用于模幂计算。在计算的开始需要一个辅助变换$x \to \tilde{x} = xR \bmod p$，在最后也需要一个辅助变换$\tilde{x} \to x = \tilde{x}R^{-1} \bmod p$。一般情况下，

这两个辅助变换对于整个计算量来说是很少的。

令 p 是大于 2 的奇数（可能是合数），定义 $n=[\log_2 p]+1$。满足 $\gcd(a,p)=1$ 的整数 a 的 Montgomery 求逆是 $a^{-1}2^n \bmod p$。算法 38 [1]是对算法 37 的修改，它用于计算 $a^{-1}2^k \bmod p$，对于某个整数 $k\in[n,2n]$。

算法 38 F_p 上的部分 Montgomery 求逆

输入：奇数 p > 2，$a\in[1,p-1]$，n = [$\log_2 p$] + 1。

输出：（x，k），其中 $n\le k\le 2n$ 且 $x = a^{-1}2^k \bmod p$。

1、u = a，v = p。

2、$x_1 = 1$，$x_2 = 0$，k = 0。

3、当 v > 0 时，重复执行

3.1 若 v 是偶数，则 v = v / 2，$x_1 = 2x_1$；

否则，若 u 是偶数，则 u = u / 2，$x_2 = 2x_2$；

否则，若 $v\ge u$，则 v = (v – u) / 2，$x_2 = x_2 + x_1$，$x_1 = 2x_1$；

否则，u = (u – v) / 2，$x_1 = x_2 + x_1$，$x_2 = 2x_2$。

3.2 k = k + 1 。

4、若 $x_1 > p$，则 $x_1 = x_1 – p$。

5、返回（x_1，k）。

算法 38 是对算法 37 的改进，主要的计算步骤 3 中 $x_1 = 2x_1$，$x_2 = 2x_2$ 这两步运算是两个算法最大的不同，将算法 37 中的除 2 改成了乘 2，所以最后的计算结果为 $x = a^{-1}2^k \bmod p$。

为了获得更高效率，通常 Montgomery 算法选择 $R = 2^{Wt} \ge 2^n$ 并采用表示法 $\tilde{x} = xR \bmod p$，这样 $\tilde{x}$ 和 $\tilde{y}$ 的 Montgomery 乘积

$$Mont(\tilde{x},\tilde{y}) = \tilde{x}\tilde{y}R^{-1} \bmod p = xyR \bmod p = zR \bmod p = \tilde{z}$$

也同时成立。算法 39 [1]是在这种表示下的求逆算法。

算法 39 F_p 上的 Montgomery 求逆算法

输入：奇数 p > 2，n = [$\log_2 p$] + 1，$R^2 \bmod p$，$\tilde{a} = aR \bmod p$，

且 gcd(a, p) = 1。

输出：$a^{-1}R \bmod p$。

1、应用算法 37 找出（x，k），其中 $x = \tilde{a}^{-1}2^k \bmod p$ 且 $n \le k \le 2n$。

2、若 k < Wt，则

2.1 $x = Mont(x, R^2) = a^{-1}2^k \bmod p$。

2.2 k = k + Wt（现在 k > Wt）。

3、$x = Mont(x, R^2) = a^{-1}2^k \bmod p$。

4、$x = Mont(x, 2^{2Wt-k}) = a^{-1}R \bmod p$。

5、返回（x）。

通过分析我们发现如果采用算法 37，并以 $R^2 \bmod p$ 和 $\tilde{a}$ 作为输入，也可以计算得到 $a^{-1}R \equiv R^2/(aR) (\bmod p)$。然而算法 39 的计算效率更高，并且 Montgomery 乘积比求逆的成本更低。

Montgomery 求逆算法和扩展的 Euclidean 算法相比，需要改变数据的表示形式，并且在后面提到的二进制域求逆运算中效率很低，很少使用，因此在本设计中不会选用 Montgomery 求逆算法。

3.5.2 二进制域求逆算法

在二进制域中的元素通常都表示成多项式的形式，在本节我们简单地用 a 来表示二进制多项式 a(x)。回忆一下二进制域 F_{2^m} 的非零元素 a 的逆是 F_{2^m} 上的唯一的一个元素 g，在二进制域 F_{2^m} 上满足 $ag = 1$，即满足 $ag \equiv 1 (\bmod f)$。这个逆元素就用符号 $a^{-1} \bmod f$ 表示，若约减多项式不会产生混淆，则可直接表示为 a^{-1}。计算这个逆元素通常有如下两个算法：

1、扩展的多项式 Euclidean 算法 [1], [8], [89], [90], [91], [92]。

2、Fermat 定理求逆算法 [93], [94], [95], [96]。

Euclidean 算法的扩展形式既可以用于素数域求逆，也可以用于二进制域的求逆运算。但在 Montgomery 算法在二进制域中的运算效率并不高，因此很少用。在前面的介绍中我们已经知道多项式的平方运算是一个线性的运算，计算效率非常高，因此利用这样的特点可以把求逆转换成求幂来运算。因此 Fermat 定理在二进制域的求逆运算中常被采用。下面简要介绍这两种算法。

多项式的扩展 Euclidean 算法

令 a 和 b 是两个不全为零的二进制多项式。a 和 b 的最大公因式表示为 $\gcd(a,b)$，它是能同时整除 a 和 b 的次数最高的二进制多项式 d。计算 $\gcd(a,b)$ 的有效算法利用了和整数 Euclidean 算法非常类似的算法：

如果 a 和 b 是两个二进制多项式，则对于所有的二进制多项式 c 都有

$$\gcd(a,b)=\gcd(b-ca,a)$$

在传统的计算二进制多项式 a 和 b（$\deg(b)\geq\deg(a)$）的最大公因式的 Euclidean 算法中，用 a 除 b 得到商 q 和余数 r，满足 $b=qa+r$，且 $\deg(r)<\deg(a)$。根据上面的定理可以得到以下结论：

$$\gcd(a,b)=\gcd(r,a)$$

因此确定 $\gcd(a,b)$ 的问题就简化为计算 $\gcd(r,a)$ 的问题，其中变量 (r,a) 的次数低于原来的变量 (a,b) 的次数。这一处理过程一直持续到其中一个变量为零为止。这时可立即得到最大公因式，因为 $\gcd(0,d)=d$。算法一定会终止，因为余数的次数是严格递减序列。而且算法是高效的，因为算法的执行最多需要 m 次除法，其中 $m=\deg(a)$。

而在 Euclidean 算法的一种变体中，每一次除法只要一步就能完成。即若 $\deg(b)\geq\deg(a)$，且 $j=\deg(b)-\deg(a)$，则一次计算 $r=b-x^{j}a$。根据上面提到的结论既可得到 $\gcd(a,b)=\gcd(r,a)$。这一处理过程重复到余数为零为止。因此

$\deg(r) < \deg(b)$，所以除的次数最多为 2m 次，$m = \max\{\deg(a), \deg(b)\}$。

二进制域中的 Euclidean 算法和素数域中的一样，可以扩展用来寻找多项式 g 和 h，使得 $ag + bh = d, d = \gcd(a,b)$。算法 40 [1]保持下列等式不变：

$$ag_1 + bh_1 = u$$

$$ag_2 + bh_2 = v$$

当 $u = 0$ 时算法终止，那时 $v = \gcd(a,b)$，并且 $ag_2 + bh_2 = d$。

算法 40 二进制多项式的扩展 Euclidean 算法

输入：非零二进制多项式 a 和 b，且 $\deg(b) \geq \deg(a)$。

输出：$d = \gcd(a,b)$，二进制多项式 g 和 h，其中 $ag + bh = d$。

1、$u = a$，$v = b$。

2、$g_1 = 1$，$g_2 = 0$，$h_1 = 0$，$h_2 = 1$。

3、当 $u \neq 0$ 时，重复执行

3.1 $k = \deg(u) - \deg(v)$。

3.2 若 $k < 0$，则 $u \leftrightarrow v$（u 和 v 的值互换），$g_1 \leftrightarrow g_2$，$h_1 \leftrightarrow h_2$，$k = -k$。

3.3 $u = u + x^k v$。

3.4 $g_1 = g_1 + x^k g_2$，$h_1 = h_1 + x^k h_2$。

4、$d = v$，$g = g_2$，$h = h_2$。

5、返回 (d, g, h)。

现在假设 f 是一个次数为 m 的既约二进制多项式，a 是次数最高为 $m-1$ 的非零多项式，显然 $\gcd(a, f) = 1$。若在上面的算法 40 中以 f 和 a 作为输入进行计算，则在步骤 3.3 就遇到 $u = 1$。此后在步骤 3.4 中多项式 g_1 和 h_1 满足 $ag_1 + fh_1 = 1$。因此，$ag_1 \equiv 1 \pmod{f}$，于是 $a^{-1} = g_1$。注意，对于确定 g_1 的计算，h_1 和 h_2 是不需

要的。在二进制域中对于 Euclidean 算法的扩展和使用方式与素数域中是完全一样的。根据这些分析，得出用于求二进制域 F_{2^m} 上的元素逆的算法 41 [1]。

算法 41 用扩展的 Euclidean 算法求 F_{2^m} 上的逆

输入：次数最高为 m – 1 的非零二进制多项式 a 和既约多项式 f。

输出：$a^{-1} \bmod f$ 。

1、$u = a$， $v = f$ 。

2、$g_1 = 1$， $g_2 = 0$ 。

3、当$u \neq 1$时，重复执行

3.1 $k = \deg(u) - \deg(v)$ 。

3.2 若$k < 0$，则$u \leftrightarrow v$（u 和 v 的值互换），$g_1 \leftrightarrow g_2$， $k = -k$ 。

3.3 $u = u + x^k v$ 。

3.4 $g_1 = g_1 + x^k g_2$ 。

4、返回（g_1）。

注意在算法 41 的步骤 3.3 和 3.4 中都含有乘以 x^k 的运算，这对于硬件实现来说要用到左移操作，因此我们称这种方法为左移算法。对于硬件实现来说，很多的文献中都采用这种方法来计算二进制域上的求逆运算。不仅因为左移操作非常容易用硬件来实现，而且步骤 3.4 中的条件判断$k < 0$实现起来也非常容易，由于是和“0”比较，因此并不需要使用比较器，只需要判断符号位即可。这是算法 41 的主要优点。算法 42 是我们在参考了文献上已有的算法 [8]的基础上加以改进后的求逆算法，这个算法可以直接用于硬件实现。

算法 42 求 F_{2^m} 上的逆的硬件实现算法

输入：次数最高为$m-1$的非零二进制多项式 a 和既约多项式 f。

输出：$a^{-1} \bmod f$ 。

1、$S=f,R=a,U=1,V=0$，（假设 S 和 R 的位数都为 m 位，对 S、V、R、U 进行初始化）。

2、$delta=0$，（$delta=\deg S-\deg R$，degS 和 degR 分别表示 S 和 R 的次数）。

3、当 i 从 1 累加到 2m，重复执行（同时判断 r_m、s_m 和 delta 这三个变量的值，一共有 5 种情，其中 r_m、s_m 分别为 S 和 R 的最高位）

3.1 若 $r_m=0$，则

$R=xR,U=(xU)\bmod F,delta=delta+1$，（R 的次数减少 1）。

3.2 若 $s_m=0$ 且 $delta\neq 0$，则

$S=xS,U=(U/x)\bmod f,delta=delta-1$，（S 的次数减少 1）。

3.3 若 $s_m=1$ 且 $delta\neq 0$，则

$S=x(S-R),V=(V-U)\bmod f,U=(U/x)\bmod f,delta=delta-1$ ，（S 的次数减少 1）。

3.4 若 $s_m=0$ 且 $delta=0$，则

$R=xS,S=R,U=(xV)\bmod f,V=U$，（前面的 4 步操作同时进行，实际上是完成了 $S=xS,R\leftrightarrow S,U\leftrightarrow V,U=(xU)\bmod f$ 这几步运算，将串行操作改成了并行操作，在没有增加任何资源的情况下加快了运算速度），$delta=delta+1$，（R 的次数减少 1）。

3.5 若 $s_m=1$ 且 $delta=0$，则

$R=x(S-R),S=R,U=(x(V-U))\bmod f,V=U$，（前面这 4 步操作也同时进行，相当于完成了 $S=S-R,V=(V-U)\bmod f,R\leftrightarrow S,U\leftrightarrow V,U=(xU)\bmod f$ 这几步运算，将串行操作改成了并行操作，在没有增加任何资源的情况下加快了运算速度），$delta=delta+1$，（R 的次数减少 1）。

4、返回（U）（最后 U 的值就是模逆的结果，即 $U=a^{-1}\bmod f$）。

与左移算法相对应，必然也会有右移算法。算法 43 [1]就是通过右移操作来完成求逆运算的。

算法 43 求二进制域 F_{2^m} 上的右移算法

输入：次数最高为 $m-1$ 的非零二进制多项式 a 和既约多项式 f。

输出：$a^{-1} \bmod f$ 。

1、$u=a$，$v=f$ 。

2、$g_1=1$，$g_2=0$ 。

3、当 $u\neq 1$ 且 $v\neq 1$ 时，重复执行

3.1 当 x 整除 u 时，重复执行

$u=u/x$，

若 x 整除 g_1，则 $g_1=g_1/x$；否则，$g_1=(g_1+f)/x$ 。

3.2 当 x 整除 v 时，重复执行

$v=v/x$，

若 x 整除 g_2，则 $g_2=g_2/x$；否则，$g_2=(g_2+f)/x$ 。

3.3 若 $\deg(u)>\deg(v)$，则

$u=u+v, g_1=g_1+g_2$；

否则，$v=u+v, g_2=g_1+g_2$ 。

4、若 $u=1$，则返回（g_1）；

否则，返回（g_2）。

Fermat 定理求逆算法

在二进制域 F_{2^m} 上的求逆算法中除了可以使用扩展的 Euclidean 算法以外，还有一种很重要的算法，Fermat 定理求逆算法。

Fermat 定理求逆算法[1]的理论基础为：

有限域 F_q（F_q既可表示素域 F_p，也可表示二进制域 F_{2^m}）的全体非零元素构成一个乘法循环群，记为 F_q^*。在 F_q^* 中存在一个元素 b，b 称为生成元，使得

$$F_q^* = \{b^i : 0 \le i \le q-2\}$$

$a \in F_q^*$ 的阶是满足 $a^t = 1$ 的正整数 t。因为 F_q^* 是循环群，所有 t 是 $q-1$ 的因子 [1]。

在以上的理论基础下给出 Fermat 定理 [93], [94], [95], [96]：

对于任意的 $a \in F_{2^m}$，有 $a^{2^m} = a$，所以 $a^{-1} = a^{2^m-2}$。

根据费马定理，对于二进制域 F_{2^m} 上的任意元素 a，有 $a^{-1} \bmod f = a^{2^m-2}$。因为 $2^m - 2 = 2 + 2^2 + 2^3 + \cdots + 2^{m-1}$，所以 $a^{-1} \bmod f = a^{2+2^2+2^3+\cdots+2^{m-1}} = a^2 \cdot a^{2^2} \cdot a^{2^3} \cdots\cdots a^{2^{m-1}}$。这种算法主要是把模逆运算转化成了模幂运算，通过上面的式子可以把模幂运算分解成为 $m-2$ 个模乘运算和 $m-1$ 个模平方运算。由于在二进制域中计算模平方运算非常方便，因此 Fermat 定理求逆算法在二进制域中被广泛使用。

费马定理和前面提到的扩展的 Euclidean 算法相比运算比较复杂，并且，我们并没有设计专门计算二进制域模平方的运算单元，无法发挥 Fermat 定理的优势。基于上述两方面原因，在本设计中并不会选用费马定理求逆算法。

3.5.3 同时支持双有限域的求逆算法

通过前面对素数域和二进制域两个域中求逆算法的分析，扩展的 Euclidean 算法可以同时应用到这两个有限域中，通过改进分别得到了支持素数域求逆的算法 37 和支持二进制域求逆的算法 43。这两个算法的计算效率都很高，更重要的是它们有很多相同的运算步骤。因此在这一部分我们把这两种算法进行合并，设计出可以同时支持双有限域的求逆算法，并对这个算法作出必要的改进，以使其更适合于硬件实现。

通过分析，算法 37 和算法 43 中都包括 4 个主要步骤，其中步骤 1、2 和 4 基本上相同，主要是在步骤 3 中的一些计算有所不同。由于在使用算法 37 时，素数域的元素要表示成二进制数的形式，而在算法 43 中二进制域的元素的多项式表示也表示成二进制数形式，因此两个有限域上的元素表达形式完全可以统一。所以步骤 3 的区别主要是由于素数域和二进制域的加、减和取模运算不同造

成的。这没有办法改进，但是可以合并两个算法的条件判断以及控制部分，使得统一的算法节省大量的操作，算法 44 体现了我们对于算法 37 和算法 43 的合并。

算法 44 同时支持素数域和二进制域的统一求逆算法

输入：模 p（素数域的模 p 或既约多项式 f)、求逆元素 a 和有限域判断变量 m（当$m=1$时表示素数域，当$m=0$时表示二进制域） 。

输出：$a^{-1} \bmod p$。

1、$u=a$， $v=p$。

2、$x_1=1$， $x_2=0$。

3、当$u\neq 1$和$v\neq 1$时，重复执行

3.1 当 u 是偶数时，重复执行

$u=u/2$，

若 x_1 是偶数，则$x_1=x_1/2$；

否则，当$m=1$时， $x_1=(x_1+p)/2$；

当$m=0$时， $x_1=(x_1\oplus p)>>1$。

3.2 当 v 是偶数时，重复执行

$v=v/2$，

若 x_2 是偶数，则$x_2=x_2/2$；

否则，当$m=1$时， $x_2=(x_2+p)/2$；

当$m=0$时， $x_2=(x_2\oplus p)>>1$。

3.3 若$u\geq v$，

则，当$m=1$时， $u=u-v, x_1=x_1-x_2$；

当$m=0$时， $u=u\oplus v, x_1=x_1\oplus x_2$。

否则，当$m=1$时， $v=v-u, x_2=x_2-x_1$；

当$m=0$时， $v=v\oplus u, x_2=x_2\oplus x_1$。

4、若$u=1$，则返回（$x_1 \bmod p$）；否则，返回（$x_2 \bmod p$）。

算法 44 的合并效率很高，没有引入任何算法 37 和算法 43 以外的运算，只是增加了一个输入参数 m，作为对两个有限域的判断。在算法 44 中，为了更为明确的区分两个域的运算，用按位异或运算符号“⊕”表示二进制域中的加法或减法运算。这在前面分析二进制域的加、减运算时已经做出分析，在二进制域加法和减法都可以用按位异或运算来完成。

上面提出的统一求逆算法仅仅是合并已有的算法，仍然存在不足，需要做出进一步的改进。首先在算法 44 的步骤 4 中 x_1 和 x_2 需要取模后再输出，即在求逆运算结束以后需要进行取模运算，这种情况非常类似于前面分析过的有限域乘法运算。我们仍然希望通过改进实现求逆运算和取模运算的同时完成，从步骤 3.3 可以看出当 $m=1$ 时会进行 $x_1 = x_1 - x_2$ 或者 $x_2 = x_2 - x_1$ 的运算，在运算之前不知道 x_1 和 x_2 哪个值大，因此计算出来的结果有可能出现负值。如果在这步计算之前先判断 x_1 和 x_2 的大小、后计算，当值为负数时立刻执行加模数 p 的取模运算，则可以保证计算结果是正值，并且保证 $x_1 \in [1, p-1]$ 或者 $x_2 \in [1, p-1]$，从而省去了输出时取模的运算，使得在做求逆运算的同时完成取模运算，这个重要改进将会明显提高求逆运算的效率。

从算法 44 中还可以看出在步骤 3.1 和 3.2 计算完成后 u 和 v 都向右移一位，缩短了其位数，但是步骤 3.3 的操作只是一个减法运算，这步运算不一定会缩短 u 或者 v 的位数，从而会增加操作步。本设计针对这一点也提出了改进。由于当需要执行 3.3 步之前，u 和 v 肯定都是奇数，因此，无论这步进行的是 $u = u - v$ 还是 $v = v - u$ 操作，计算结果都是一个偶数，因此下一步必将执行 3.1 和 3.2 这两步中的一步，即判断移位操作。本设计对步骤 3.3 计算出来的结果直接进行判断移位操作，使它们在一步里完成，从而缩短了操作步的数量，提高了运算速度。

算法 45 体现了上面分析的两点重要改进：

算法 45　改进后的同时支持素数域和二进制域的统一求逆算法

输入：模 p（素域的模 p 或既约多项式 f）、求逆元素 a 和有限域判断变量 m（当 $m=1$ 时表示素域，当 $m=0$ 时表示二进制域）。

输出：$a^{-1} \bmod p$。

1、$u = a$，$v = p$。

2、$x_1 = 1$，$x_2 = 0$。

3、当$u \neq 1$和$v \neq 1$时，重复执行

3.1 当 u 是偶数时，重复执行

$u = u/2$，

若 x_1 是偶数，则 $x_1 = x_1/2$；

否则，当$m = 1$时，$x_1 = (x_1 + p)/2$；

当$m = 0$时，$x_1 = (x_1 \oplus p) >> 1$。

3.2 当 v 是偶数时，重复执行

$v = v/2$，

若 x_2 是偶数，则 $x_2 = x_2/2$；

否则，当$m = 1$时，$x_2 = (x_2 + p)/2$；

当$m = 0$时，$x_2 = (x_2 \oplus p) >> 1$。

3.3 若$u \geq v$，

则，当$m = 1$时，$u = (u - v)/2$；

若$x_1 > x_2$，则

当 x_1 和 x_2 奇偶性相同时，$x_1 = (x_1 - x_2)/2$；

当 x_1 和 x_2 奇偶性不同时，$x_1 = (x_1 - x_2 + p)/2$；

否则，

当 x_1 和 x_2 奇偶性相同时，$x_1 = (x_1 - x_2)/2 + p$；

当 x_1 和 x_2 奇偶性不同时，$x_1 = (x_1 - x_2 + p)/2$；

当$m = 0$时，$u = (u \oplus v) >> 1$；

当 x_1 和 x_2 奇偶性相同时，$x_1 = (x_1 \oplus x_2) >> 1$；

当 x_1 和 x_2 奇偶性不同时，$x_1 = (x_1 \oplus x_2 \oplus p) >> 1$。

否则，当 $m = 1$ 时，$v = (v - u)/2$；

若 $x_2 > x_1$，

当 x_1 和 x_2 奇偶性相同时，$x_2 = (x_2 - x_1)/2$；

当 x_1 和 x_2 奇偶性不同时，$x_2 = (x_2 - x_1 + p)/2$；

否则，

当 x_1 和 x_2 奇偶性相同时，$x_2 = (x_2 - x_1)/2 + p$；

当 x_1 和 x_2 奇偶性不同时，$x_2 = (x_2 - x_1 + p)/2$；

当 $m = 0$ 时，$v = (v \oplus u) >> 1$；

当 x_1 和 x_2 奇偶性相同时，$x_2 = (x_1 \oplus x_2) >> 1$；

当 x_1 和 x_2 奇偶性不同时，$x_2 = (x_1 \oplus x_2 \oplus p) >> 1$。

4、若 $u = 1$，则返回（x_1）；否则，返回（x_2）。

改进后的算法 45 中的步骤 3.3 和算法 44 相比较，在条件判断上复杂了一些，但是在减法运算的同时进行了移位操作，这就使得步骤 3.1、3.2 和 3.3 这三步运算的每一步都至少向右移了一位，对于 m 位的运算，保证在 $2m$ 步运算内输出结果，在运算速度上比原来的算法提高了一倍。

首先，本设计在只支持一种有限域的求逆算法的基础上，提出了一种同时支持素数域和二进制域的求逆算法。把统一求逆算法和只支持一种有限域的求逆算法相比较可以看出除了增加了一个有限域判断变量 m 外，并没有增加其它变量，节省了运算资源。其次，原算法中对于素数域的计算结果需要取模后再输出，在素数域上的取模运算就是通过加 p 或者减 p 的运算使得最终的结果在 1 至 $(p-1)$ 这个范围内。按照原算法的计算过程，最后的结果是不可知的，需要专门设计取模算法。即使设计了取模算法，取模的过程所耗费的时钟周期也是不固定的，可能需要很多个时钟周期才能完成取模运算。本设计改进了这个取模的过程，在算

法 45 的步骤 3.3 中计算出结果后立刻对结果取模，这样保证了最后计算结果在 1 至 $(p-1)$ 之间，同时也为后面进一步提高速度提供了条件。第三，统一算法在执行步骤 3.3 时变量 u 和 v 同为奇数，因此它们的差肯定是一个偶数，下一步必定会进行除以 2 的运算，因此本设计在原算法的基础上对步骤 3.3 增加了除以 2 的操作，这样可以只用 $2m$ 个时钟周期的时间来计算 m 位的求逆结果，这在已知的有限域模逆算法中计算效率是最高的。当然，这种改进同时也增加了对于 x_1 和 x_2 两个变量的判断过程，但这并不会对运算速度造成很大的影响，同时对于运算资源的增加也非常有限。算法 45 就是我们最终所使用的可以同时支持素数域和二进制域的求逆算法。

这个算法的计算量仅仅依赖输入数据，和存储器的位长没有关系，因此硬件结构可以设计得很灵活。输入数据主要是求逆元素 a 和模 p，当 a 为 m 位的二进制数时，模 p 通常为 $m+1$ 位，计算结束以后存储这两个数据的寄存器的结果应为“1”，由于改进后的算法保证每个循环运算寄存器 u 和 v 至少有一个向右移一位，因此最多需要 $2m-1$ 次运算就可完成求逆。在这 $2m-1$ 次运算中素数域和二进制数域所做的运算是截然不同的。对于素数域的求逆来说，一次循环最多需要完成两次比较运算、一次加法运算、一次减法运算和两次移位运算，最少只需完成一个加法运算和两次移位运算。根据输入数据的不同，完成的运算也不同。二进制域上的求逆比素数域要快得多，同样是 $2m-1$ 次运算，在每次运算中最多需要完成两次比较运算、两次按位异或运算和两次移位运算，最少只需完成一次按位异或运算和两次移位运算。虽然在两个域中都需完成比较运算，但是在后面的硬件设计中我们将提出一种非常简单的设计，以提高比较运算的速度。这样由于按位异或要比加法或减法运算快得多，因此二进制域上的求逆运算也就比素数域上的求逆运算速度要快很多。

在这里需要特别说明，算法 45 不仅仅只能进行求逆运算，如果将算法中的步骤 2 改写成 $x_1 = b$， $x_2 = 0$，其它的步骤保持不变，则输出的计算结果为 $b/a \bmod p$，这一点可以通过 Euclidean 传统算法的证明得到验证，在这里就不详细给出证明过程。但是我们最后得到的结论是算法 45 不仅可以计算有限域上的求逆运算，还可以完成任意两个非零元素的除法运算，这使得算法 45 在有限域

运算中的使用上更加灵活。

有限域求逆算法的设计是点乘运算算法设计最为重要的一个环节，纵观国内外发表的文献，绝大部分成果都是出自求逆运算设计。求逆运算普遍认为比加法、减法和乘法运算要复杂得多，这里的“复杂”主要是指运算时间长、消耗资源大。因此很多文献提出了多种算法简化求逆运算或者通过变换有限域从而减少求逆算法的次数。在这一部分我们详细地介绍了对于求逆算法的选择以及改进。通过我们大量的研究和设计，最终完成了算法 45 的改进。这个算法在运算速度上和乘法算法相同，硬件资源可以和其它运算共享。

算法 45 集中体现了我们对于有限域求逆算法的理解和改进。改进的具体内容总结如下：

1、在素数域和二进制域上的求逆算法都选择了扩展的 Euclidean 算法，并将两种算法合二为一。这是非常重要的创新，在目前已经查到的国内外各种文献中我们是第一个提出这种思想的。

2、扩展的 Euclidean 算法是对最后的输出结果取模，而我们的设计将取模运算放到了求逆运算的中间步骤当中，即同时完成取模和求逆运算。改进后的算法最多需要 $2m$ 个时钟周期的时间就可以完成 m 位的求逆运算，并且输出取模以后的结果。

3、本设计利用 u 和 v 两个寄存器里的数值在运算中的特点，对步骤 3.3 计算出来的结果直接进行判断移位操作，使它们在一步里完成。以增加对 x_1 和 x_2 的判断为代价，但是达到了缩短操作步的数量，提高了运算速度的效果。

4、在已有的求逆算法的基础上加进了除法算法。用除法运算代替求逆运算和乘法运算将会提高 ECC 点乘运算的速度。

3.6 同时支持双有限域运算的混合算法

通过前面对于素数域和二进制域上加法、减法、乘法、平方、求逆以及除法等各种运算的分析，我们已经可以设计出一个同时支持双有限域运算的混合算法。

算法 15 和算法 16 可以计算素数域上的加、减法，算法 17 可以计算二进制域上的加、减法。算法 29 和算法 30 分别是计算素数域和二进制域乘法的算法，

而且这两个算法在计算乘法的过程中，在不增加运算周期的情况下就完成取模运算，效率很高。对于素数域和二进制域的求逆和除法运算，算法 45 可以完成，其输出的结果已经是取模后的最后结果。在完成有限域的平方运算时所有的算法都需要额外的增加取模运算才能实现。但是我们所选用的乘法算法和求逆算法，它们的计算结果已经是取模以后的结果了，再加上算法 45 是以扩展的 Euclidean 算法为基础，这样二进制域的快速平方运算不能够很好的发挥它的优点。因此我们没有使用额外的算法来单独计算有限域上的平方，而是用有限域上的乘法算法来计算平方运算。这样，最后的混合算法是由算法 15、16、17、29、30 和算法 45 共同组成的。

算法 46 给出了混合算法，由于这个算法所支持的运算种类很多，因此我们增加了一个输入变量 mode，来表示所做的计算的种类。具体的对应关系如下表。

表 3.5：算法 46 可支持的运算

mode	运算种类
0	初始化
1	素数域加法
2	素数域减法
3	二进制域加减法
4	素数域乘法或平方
5	二进制域乘法或平方
6	素数域求逆
7	素数域除法
8	二进制域求逆
9	二进制域除法

算法 46 同时支持双有限域运算的混合算法

输入：模 p（素域的模 p 或既约多项式 f）、有限域上的非零元素 a 和 b，运算模式 mode。

输出：运算结果 c。

1、若 mode = 1，则

1.1 $c = a + b$。

1.2 若 $c \geq p$，则 $c = c - p$。

1.3 返回（c）$(c = (a + b) \bmod p)$。

2、若 mode = 2，则

1.1 $c = a - b$。

1.2 若 $c < 0$，则 $c = c + p$。

1.3 返回（c）$(c = (a - b) \bmod p)$。

3、若 mode = 3，则

3.1 $c = a \oplus b$。

3.2 返回（c）$(c = (a \oplus b) \bmod f)$。

4、若 mode = 4，则

4.1 $c = 0$。

4.2 重复执行

4.2.1 若 $a_0 = 1$，则 $c = c + b$。

4.2.2 若 $c \geq p$，则 $c = c - p$。

4.2.3 $b = b << 1$。

4.2.4 若 $b \geq p$，则 $b = b - p$。

4.2.5 $a = a >> 1$。

4.2.6 若 $a = 0$，则执行步骤 4.3。

4.3 返回（c）$(c = a \cdot b \bmod p)$。

5、若 mode = 5，则

5.1 $c = 0$。

5.2 重复执行

5.2.1 若 $a_0 = 1$，则 $c = c \oplus b$。

5.2.2 $b = b << 1 \bmod f$。

5.2.3 $a = a >> 1$。

5.2.4 若 $a = 0$，则执行步骤 5.3。

5.3 返回（c）$(c = a \cdot b \bmod f)$。

6、若 mode = 6、7、8 或 9，则

6.1 $u = a$， $v = p$。

6.2 若 mode = 6 或 mode = 8 则， $x_1 = 1$；

否则， $x_1 = b$；

$x_2 = 0$。

6.3 当 $u \neq 1$ 和 $v \neq 1$ 时，重复执行

6.3.1 当 u 是偶数时，重复执行

$u = u/2$，

若 x_1 是偶数，则 $x_1 = x_1/2$；

否则，当 m = 6 或 7 时， $x_1 = (x_1 + p)/2$；

当 m = 8 或 9 时， $x_1 = (x_1 \oplus p) >> 1$。

6.3.2 当 v 是偶数时，重复执行

$v = v/2$，

若 x_2 是偶数，则 $x_2 = x_2/2$；

否则，当 m = 6 或 7 时， $x_2 = (x_2 + p)/2$；

当 m = 8 或 9 时， $x_2 = (x_2 \oplus p) >> 1$。

6.3.3 若$u \geq v$，

则，当 m = 6 或 7 时，$u = (u - v)/2$；

若$x_1 > x_2$，则

当 x_1 和 x_2 奇偶性相同时，$x_1 = (x_1 - x_2)/2$；

当 x_1 和 x_2 奇偶性不同时，$x_1 = (x_1 - x_2 + p)/2$；

否则，

当 x_1 和 x_2 奇偶性相同时，$x_1 = (x_1 - x_2)/2 + p$；

当 x_1 和 x_2 奇偶性不同时，$x_1 = (x_1 - x_2 + p)/2$；

当 m = 8 或 9 时，$u = (u \oplus v) >> 1$；

当 x_1 和 x_2 奇偶性相同时，$x_1 = (x_1 \oplus x_2) >> 1$；

当 x_1 和 x_2 奇偶性不同时，$x_1 = (x_1 \oplus x_2 \oplus p) >> 1$。

否则，当 m = 6 或 7 时，$v = (v - u)/2$；

若$x_2 > x_1$，

当 x_1 和 x_2 奇偶性相同时，$x_2 = (x_2 - x_1)/2$；

当 x_1 和 x_2 奇偶性不同时，$x_2 = (x_2 - x_1 + p)/2$；

否则，

当 x_1 和 x_2 奇偶性相同时，$x_2 = (x_2 - x_1)/2 + p$；

当 x_1 和 x_2 奇偶性不同时，$x_2 = (x_2 - x_1 + p)/2$；

当 m = 8 或 9 时，$v = (v \oplus u) >> 1$；

当 x_1 和 x_2 奇偶性相同时，$x_2 = (x_1 \oplus x_2) >> 1$；

当 x_1 和 x_2 奇偶性不同时，$x_2 = (x_1 \oplus x_2 \oplus p) >> 1$。

6.4 若$u = 1$，则$c = x_1$；否则，$c = x_2$，

返回（c）（当 mode = 6 时，$c = a^{-1} \bmod p$；当 mode = 7 时，$c = b/a \bmod p$；当 mode = 8 时，$c = a^{-1} \bmod f$；当 mode = 9 时，$c = b/a \bmod f$）。

算法 46 的设计成功标志着我们对于 ECC 点乘算法设计的成功，具有决定性的意义，它也是整个设计中的一个非常重要的阶段性成果。我们将以这个算法为依据设计我们的硬件电路，从而完成整个点乘电路的设计工作。算法 46 体现了我们对于有限域运算的独到理解，融入了我们全部工作心血，体现了我们的设计思想。算法 46 和已经发表过的众多算法相比有七点改进：

1、增加了一个运算模式变量 mode，从而把素数域和二进制域上的各种运算完全统一起来。

2、在算法 46 的输入和输出中已经将素数域的元素和二进制域的元素都统一成二进制数，在全部的运算过程中不需要区分这两种有限域，使得各个运算输入和输出数据无差别。即运算资源可以在两个有限域中共享，节省了运算资源。

3、算法 46 的运算非常灵活，无论是素数域的模数 p 还是二进制域中的既约多项式 f(x)，都由外部输入，可以任意选取。

4、算法在保证运算效率的前提下兼容性很强。由于在乘法和求逆运算中对于移位方式的优化设计，使得运算系统对于所输入的数据向下兼容，即对于 m 位的运算系统，输入的数据可以为不超过 m 位的任意位数据。同时运算时间不是由硬件结构的存储器位数决定，而是由输入数据本身的位数决定。我们的这一设计不仅对于算法计算效率的提高至关重要，而且为后面的硬件设计提供了便利。

5、由于算法 46 在计算有限域乘法、求逆和除法运算时，它们的计算结果都不需要进行取模运算，已经是最终的结果，因此在算法 46 中没有取模运算单元。考虑到这一点，我们没有专门设计有限域平方运算算法，而是用乘法算法来计算平方运算。这也节省了很多资源，同时没有降低运算效率。

6、从计算效率上看，完成素数域加减法，需要 2 个运算周期；完成二进制域加减法，需要 1 个运算周期。如果输入的元素为 m 位的二进制数，计算素数

域乘法或平方，最多需要 2m 个运算周期；计算二进制域的乘法或平方，最多需要 m 个运算周期。对于 m 位的操作数，素数域和二进制域上的求逆和除法的计算最多需要 2m 个运算周期。保证各个运算效率最高。

7、算法 46 的控制单元经过精心设计，将乘法和求逆运算高效的转化成加法和减法运算，保证了所有运算资源共享。

3.7 本章小结

本章我们详细地介绍了素数域和二进制域上的算术运算，并且针对椭圆曲线加密算法所涉及的有限域乘法运算、求逆运算做出了创新性改进。最终完成了同时支持双有限域运算的混合算法（算法 46）的设计。这个算法的设计成功，标志着我们对于有限域运算的理解上升到一个新高度，同时这个算法也是我们进行硬件设计的重要基础。在下一章我们将重点讲述如何把算法 46 映射到硬件上，如何设计出高效的点乘运算电路模块。

第四章　点乘运算模块的电路设计

点乘运算是 ECC 运算的核心，它直接影响着椭圆曲线加密系统在做加、解密时的运算速度。这一点在本文前面的章节已经做出了比较详细的介绍。在这里再简单的回忆一下椭圆曲线点乘的定义：椭圆曲线上的点乘运算可以表示为 kP，其中 k 是一个整数，P 是定义在有限域 F_p 或 F_{2^m} 上的椭圆曲线 E 上的一个点。点乘 kP 的计算公式为：

$$\underbrace{P+P+\cdots+P}_{k}=kP$$

需要强调的是在这里所讨论的点乘不依赖于任何具体的曲线结构，它是适用于任何椭圆曲线的，因此具有普遍意义。所以任何的椭圆曲线加密算法都要进行点乘运算。现在需要考虑的问题是当随机选择一个整数 k 时，如何设计高性能的硬件电路来实现点乘运算 kP 的全过程。

在给出具体的设计方案之前，先提出一些在加密算法硬件设计过程中不得不考虑的问题。好的设计强调对目标平台、操作环境、性能和安全要求的全面了解。

1、成本。成本总是硬件设计者考虑的一个重要问题，它由下面所有的标准来驱动。

2、硬件和软件。是否一定需要硬件来实现，用软件来实现是否可以？在 ECC 加密算法的设计中，用软件来实现的很多，但是软件实现的速度明显比硬件慢，而且受到具体应用环境的制约。在许多要求高速实现加密解密的地方通过硬件来完成 ECC 的加密解密过程是唯一的选择。当然，如果对速度要求并不是很高的地方，选择软件实现之上再用硬件加速器的设计也是合理的。我们设计的硬件也完全适用于这种情况。

3、吞吐量。椭圆曲线加密算法的应用范围是非常广泛的，安装在服务器上的装置有可能要求在每秒钟执行上百或上千次椭圆曲线操作。综合考虑了这些情况以后，在后面的硬件设计中提出了两个设计方案：一种偏重于高速加密；另一种则强调经济性，面积小。

4、复杂性。装置实现的层次越多，其电路将越复杂，这使得它在定制的 VLSI 装置上消耗更多的硅原材料或者成为一个更大的 FPGA，其结果便是更高的成

本。这是任何一个硬件设计所必须考虑的问题，本设计将点乘运算分为三级，每一级独立设计，三级模块互相调用，结构简单。

5、适应性。相关问题包括装置在二进制域和素数域的曲线上执行计算的能力。同时支持双有限域运算的混合算法（算法 46）保证了设计的计算能力，并且具有很好的适应性。

6、算法的灵活性。很多加密协议要求在每一次会话基础上协商加密算法。可重构的硬件实现可以保证这种协议的实现，并且几乎不影响系统的性能。算法 46 强大的功能同样可以保证这一点。对于外部的输入数据 k，算法 46 可根据其位数的长短自行调整运算的次数，保证最为高效的运算。这一点在上一章已经作出了详细的介绍。

7、功耗。依赖于装置的操作环境，功耗可能是也可能不是一个主要的问题。例如，尽管服务器能够承担起更多的功耗，但是非接触式智能卡在加密操作方面仍受到限制。降低功耗在电路的底层模块中作为重点来设计。我们所设计的电路的功耗在后面会给出详细的数据。

8、安全性。安全性在任何加密算法的硬件设计中都是需要重点考虑的一部分。若装置仅被设计用来执行点加和倍点，则在点乘 kP 过程中它将被与随机数 k 相关联的比特位激活。若没有精心的设计的整体结构，则 k 的比特位可能会因旁门攻击而泄露。在装置的操作环境中，应该考虑到基于时限、能量分析和电磁辐射等的攻击。本设计在计算点加和倍点运算时与 k 的比特位无关，具有足够的安全性。

9、整体系统结构。若在整个系统中，微处理器有足够的空闲周期来处理上述有限域算术的协议功能，则根据其它标准，有理由设计出仅供有限域算术使用的装置。这正是本设计的一大特色，本设计对于点乘运算的电路结构分为三级，第三级为有限域运算单元，它不仅适用于点乘运算，而且可以满足任意有限域的运算，完全可以作为其它有限域相关设计的运算模块来使用。

10、实现平台。在定制的 VLSI、门阵列或 FPGA 等平台上均能实现。相对于 VLSI 和门阵列装置，FPGA 的每个单元成本比较高，而 VLSI 实现的设计成本也是比较高的。本设计在 VLSI 和 FPGA 这两种通用的实现平台上都作了实现。后面将给出具体的实现结果。

11、可伸缩性。若期望装置能够提供各种不同级别的安全，则必须设计出底层的有限域处理器，以适应不同的域大小。作为本设计电路结构的第三层模块有限域运算处理单元与算法 46 相对应，这也是本设计最大的创新点，很好的体现了可伸缩性的特点。

通过上面这 11 个问题的回答，也概括了硬件设计的特点。下面就给出具体的硬件设计结构。

4.1 节将讲述点乘运算的结构以及实现点乘模块的电路结构。4.2 节介绍点乘第一级模块的硬件结构。4.3 节讲述点乘的第二级模块点加和倍点模块的电路结构。4.4 节介绍点乘第三级模块有限域运算模块的电路结构。4.5 节在总结创新点的基础上对本章做出小结。

4.1 点乘的运算结构及其电路结构

由于点乘运算相当复杂，因此必须了解这个运算的具体结构，在根据运算的特点来设计我们的电路结构。

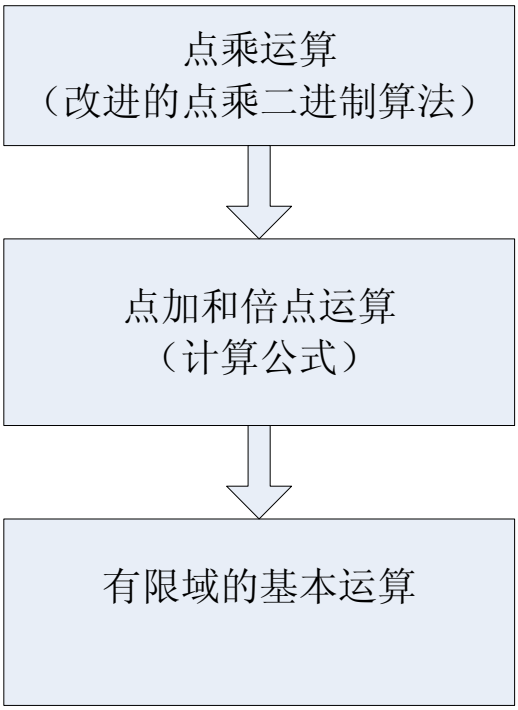

图 4.1　椭圆曲线点乘的运算结构图

从图 4.1 可以看出，椭圆曲线的点乘运算由上到下共分为 3 级来完成的：

第一级：做点乘运算，我们所采用的算法就是在前面介绍过的算法 3，算法 3 是我们在总结已有的算法的基础上做出改进以后的算法，非常适合于硬件实现。

第二级：由于在第一级的运算中，把点乘运算分解成了有限域上的点加和倍点运算，因此在第二级运算主要做这两个运算。点加和倍点运算很简单，在素数域和二进制域都有相应的计算公式。

第三级：第三级运算是点乘运算最为底层的一级，它是上两层的基础，同时也是最为复杂的一级运算。在这一级运算中将完成有限域上的所有运算，它们包括加法、减法、乘法、平方、求逆和除法运算。为此我们在上一章专门讨论了有限域上的运算，同时我们在做这个设计时也花费了大量的时间来改进每一种有限域运算的算法，最终提出了可以支持双域的有限域混合算法。混合算法可以完成素数域和二进制域双域上的任何运算，而且适合于硬件实现，计算效率高。

针对点乘运算的结构我们在硬件设计上分为 3 个模块，它们和点乘运算的 3 级运算一一对应，如图 4.2：

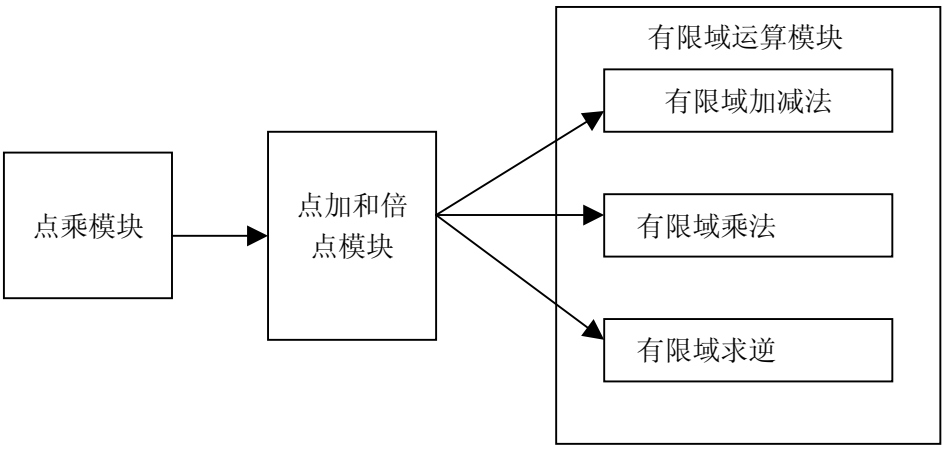

图 4.2　点乘硬件实现的电路结构

从图中可以看出硬件的分级和算法完全一致。顶层模块为点乘模块，它在运算过程中会调用第二层模块。第二层模块通过外部输入变量来判断所做的运算，它可以完成素数域和二进制域上的点加和倍点运算，第二层模块在运算时又会调

用第三层模块。第三层模块为有限域的运算模块。这个模块既可以做为 ECC 点乘的底层模块使用，也可以做为其它有限域算法的运算模块使用。很多文献中都是用一个大的控制模块来控制整个点乘运算，这种方法有两个缺点：一是会浪费时钟周期；二是使电路变得复杂。因此在本设计的电路中三层模块独立完成自己的运算功能，模块之间只通过握手信号进行数据传输，且握手信号全部与时钟同步，使整个电路非常简单。而且由于各级模块都是独立的，因此增加了电路的可移植性。下面就具体介绍各级模块的电路实现。

4.2 点乘模块的硬件结构

点乘运算作为第一级运算，并不十分复杂。在前面我们介绍了计算点乘的算法 1 和算法 2，它们是计算点乘的二进制算法，非常适合于硬件实现。这个算法的核心是把 kP 中的 k 用二进制表示，然后根据公式把 kP 的计算分解成点加和倍点运算。当需要计算点加和倍点运算时将会调用第二级运算。因此在这一级，主要是对控制电路的设计。无论是算法 1 还是算法 2，对于控制条件的判断很固定，都是将 k 表示成 $k=(k_{m-1},\cdots,k_2,k_1,k_0)_2$，然后通过 m 次的运算，最终完成点乘运算。如果按照算法 1 和算法 2 来设计这一级电路，最后的电路结构将严重依赖于 k 的二进制数表示的长度 m，这将影响系统的灵活性。针对这个问题，我们又对算法 1 和算法 2 作出了调整，使得这一级电路不依赖于 m 的取值，提出了算法 3。

算法 3 的改进主要体现在步骤 2.3 和 2.4，通过对 k 的右移操作和对 k 是否为“0”的判断，来控制点加和倍点的运算次数。当 k 的值为“0”时算法 3 将会自动终止，节省了运算时间。这种控制比固定的执行 m 次操作要灵活得多，使得算法 3 可以实现任意位长的点乘运算。在电路结构上可以分析出这部分需要的硬件仅仅为一个 m 位的移位器和相应的控制电路。具体的电路结构如图 4.3（图中并没有给出时钟信号以及握手信号）。

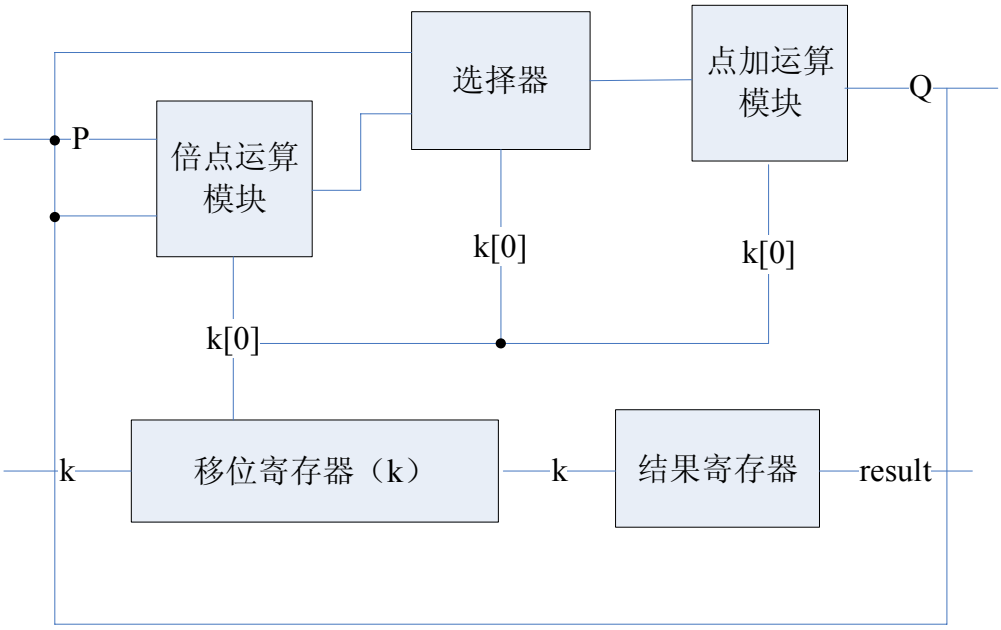

图 4.3　点乘第一级模块电路结构图

当算法 3 执行到步骤 2.1 和 2.2 时将会调用系统的第二级模块，点加和倍点模块。下面就详细介绍第二级模块。

4.3 点加和倍点模块的硬件结构

对于素数域和二进制域的点加和倍点来说，它们都有固定的运算公式：

在素数域中，点加和倍点的计算公式分别为：

对于任意两点 P,Q，$P,Q\in E(F_p)$，且 $P\neq\pm Q$。令 $P=(x_1,y_1)$，$Q=(x_2,y_2)$，$P+Q=(x_3,y_3)$，那么有：

$$x_3=(\frac{y_2-y_1}{x_2-x_1})^2-x_1-x_2$$

$$y_3=(\frac{y_2-y_1}{x_2-x_1})(x_1-x_3)-y_1$$

对于任意的点 P，$P=(x_1,y_1)\in E(F_p)$，要求 $P\neq -P$，那么 $P+P=2P=(x_3,y_3)$，那么有：

$$x_3 = (\frac{3x_1^{\ 2} + a}{2y_1})^2 - 2x_1$$

$$y_3 = (\frac{3x_1^{\ 2} + a}{2y_1})(x_1 - x_3) - y_1$$

二进制域的点加和倍点公式为：

对于任意两点P,Q，$P,Q \in E(F_{2^m})$，且$P \neq \pm Q$。令$P = (x_1, y_1)$，$Q = (x_2, y_2)$，$P + Q = (x_3, y_3)$，那么有：

$$x_3 = (\frac{y_1 + y_2}{x_1 + x_2})^2 + \frac{y_1 + y_2}{x_1 + x_2} + x_1 + x_2 + a$$

$$y_3 = (\frac{y_1 + y_2}{x_1 + x_2})^2 (x_1 + x_3) + x_3 + y_1$$

对于任意点P，$P = (x_1, y_1) \in E(F_{2^m})$，要求$P \neq -P$，那么$P + P = 2P = (x_3, y_3)$，那么有：

$$x_3 = x_1^{\ 2} + \frac{b}{x_1^{\ 2}}$$

$$y_3 = x_1^{\ 2} + (x_1 + \frac{y_1}{x_1})x_3 + x_3$$

对于上面两个公式里的参数 a 和 b 分别是二进制域椭圆曲线方程中的参数。

通过上面的 4 组公式可以看出在这一级电路中，主要是设计控制电路来完成点加和倍点的运算，同时增加适当的寄存器来存储中间结果。而具体的运算将会调用第三级模块有限域运算模块来完成。因此这一级电路的设计重点就是控制电路。

在具体的设计中应当考虑如下 3 个问题：

（1）点加和倍点运算中一共包含了哪几种运算，每一种运算共执行了几次。

（2）对于有限域的运算，加、减法只需一到两个时钟周期，而乘法、求逆和除法的运算时间将会远远超过加、减法的运算时间，这点在设计中也要考虑。

（3）当所设计的硬件电路强调经济性，追求面积最小化时，第三级模块中的每一种运算电路只能有一个，而且要尽量减少用于存储中间变量的寄存器数量，因此需要协调好各个运算的先后顺序；而当所设计的电路强调高性能，要求运算速度达到最快时，第三级模块中的每一种运算电路需要几个最为合理。即所

设计的电路是突出经济性还是突出性能。

图 4.4 是根据上面给出的 4 个公式做出的 4 个运算的流程图，从图中可以分析出每一钟运算所需要的总时间以及需要增加的寄存器的数量。同时也可以找到上面提出的三个问题的答案。

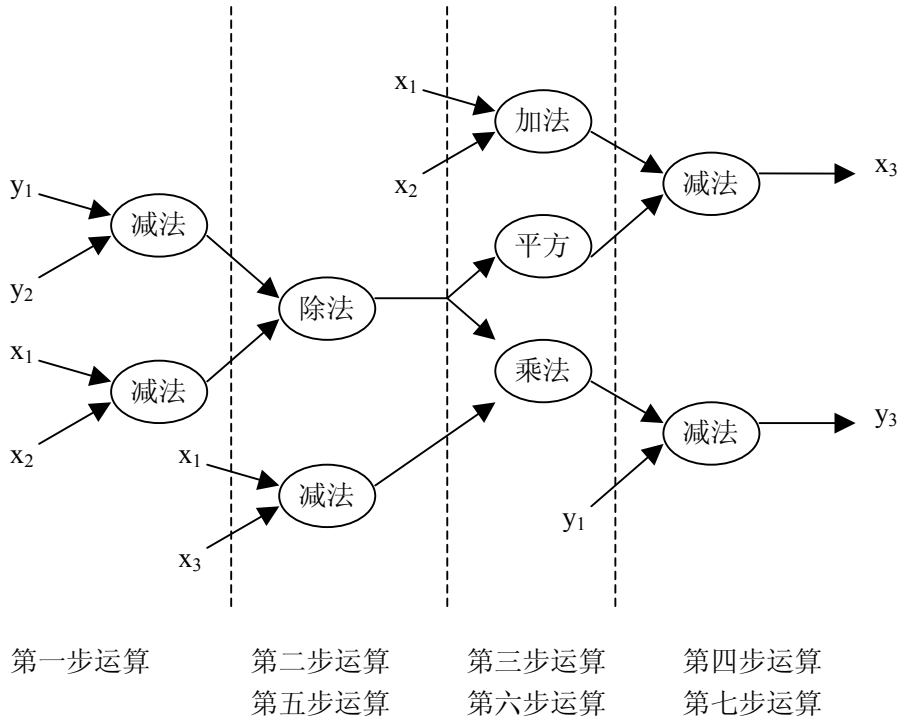

图 4.4（a） 素数域点加运算流程图

图 4.4（a）为素数域点加运算的流程图，从图中可以看出在完成点加运算时，计算 x_3 最多需要 4 个运算步骤，最长的路径为减法→除法→平方→减法（以后描述运算步骤时只给出最长的路径）。当 x_3 的结果计算出来以后将从第二步开始继续计算 y_3，还需要减法→乘法→减法 3 个步骤。这样完成整个素数域点加运算的最长路径为 7 个运算步骤：减法→除法→平方→减法→减法→乘法→减法。除法、平方和乘法这三步运算将花费大量的运算时间。其中在第一步需要做两个减法运算，如果最终的设计以节省面积为主，即每一种运算模块只设计一个，那么计算第一步的运算时间将会延长一倍，不过由于是一个减法运算，不会对最后的计算总时间造成很大影响。在第三步虽然既有平方运算，又有乘法运算，但

是这两个运算不可能同时进行。因为第二步中的减法运算的输入为 x_1 和 x_3，其中 x_3 在平方第三步的平方运算之后才能计算出来，因此第三步运算中的乘法一定在平方运算之后。第一步运算需要完成两个减法运算，它们的输出结果都是第二步运算中除法运算的输入数据，因此如果在设计中只设计了一个减法运算单元，则要增加一个寄存器来存储第一个减法运算的输入结果，同时也会使运算时间增加一个时钟周期。综上所述，对于素数域点加的设计非常灵活，根据实际情况的要求，既可以设计出高速的点加模块，也可以设计出面积小、经济实用的模块。由于在这一级电路中包括素数域点加和倍点运算以及二进制域点加和倍点运算 4 种运算，而且任意两种运算不可能同时进行，因为存储中间计算结果的寄存器可以共享。

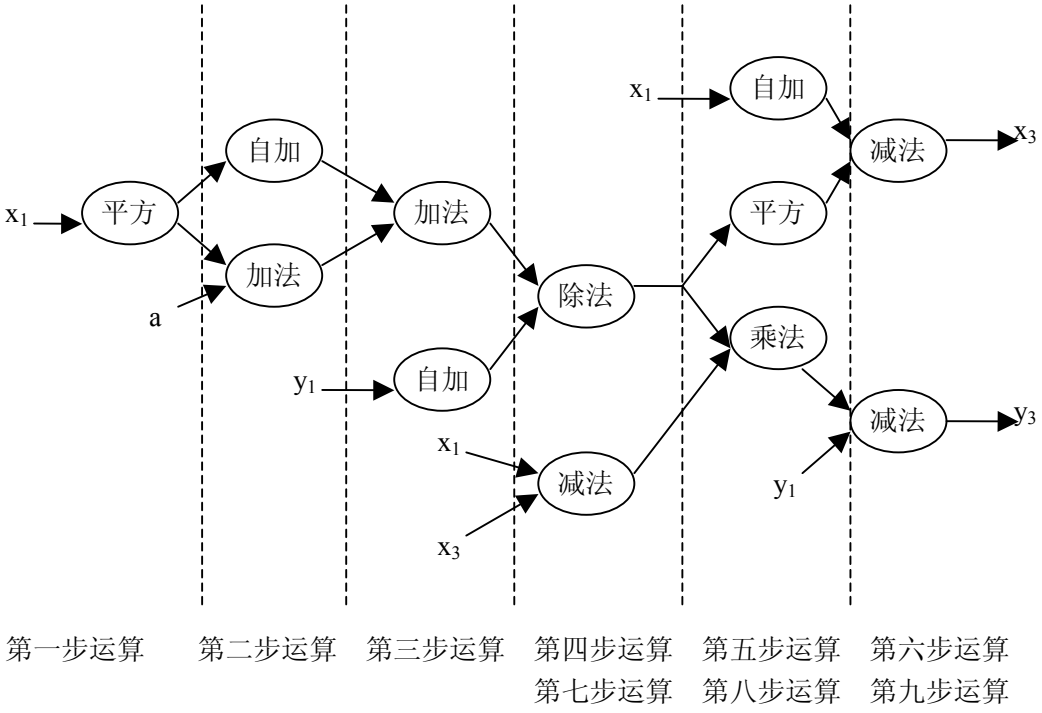

图 4.4（b） 素域倍点运算流程图

图 4.4（b）为素数域倍点运算的流程图，素数域倍点相对复杂，一共需要 6 个运算步骤才可以完成 x_3 的运算：平方→加法→加法→除法→平方→减法。在计算完 x_3 以后还需要 3 个运算步骤减法→乘法→减法完成 y_3 的运算。这样，

完成素数域倍点的计算的最长路径为：平方→加法→加法→除法→平方→减法→减法→乘法→减法，一共 9 步。其中在流程图中出现的“自加”运算就是输入数据自己和自己相加，和加法运算完全一样。最多需要一个寄存器存储中间结果。

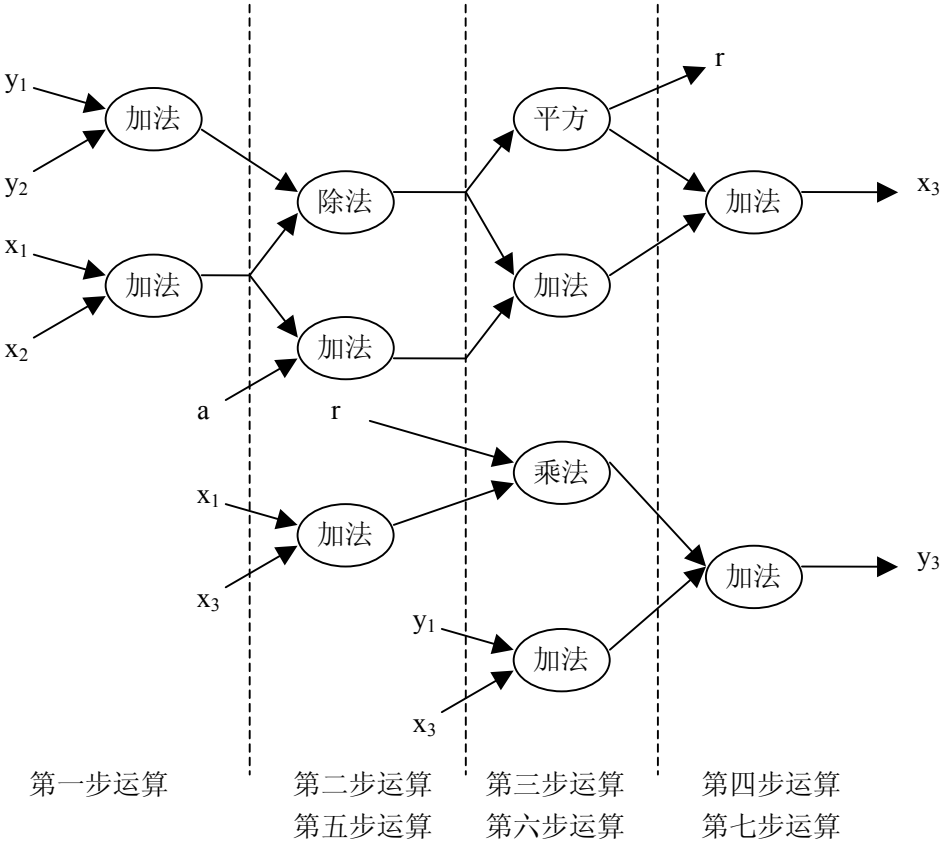

图 4.4（c） 二进制域点加运算流程图

图 4.4（c）是二进制域点加运算流程图。经过 4 步运算：加法→除法→平方→加法既可完成对于 x_3 的运算。同时可以计算出一个中间结果 r，在后面计算 y_3 时还要调用这个中间结果。当 x_3 计算完成后再增加 3 个步骤：加法→乘法→加法就可以完成 y_3 的计算。计算二进制域点加运算一共需要 7 个运算步骤：加法→除法→平方→加法→加法→乘法→加法。还需要增加一个额外寄存器存储中间结果。

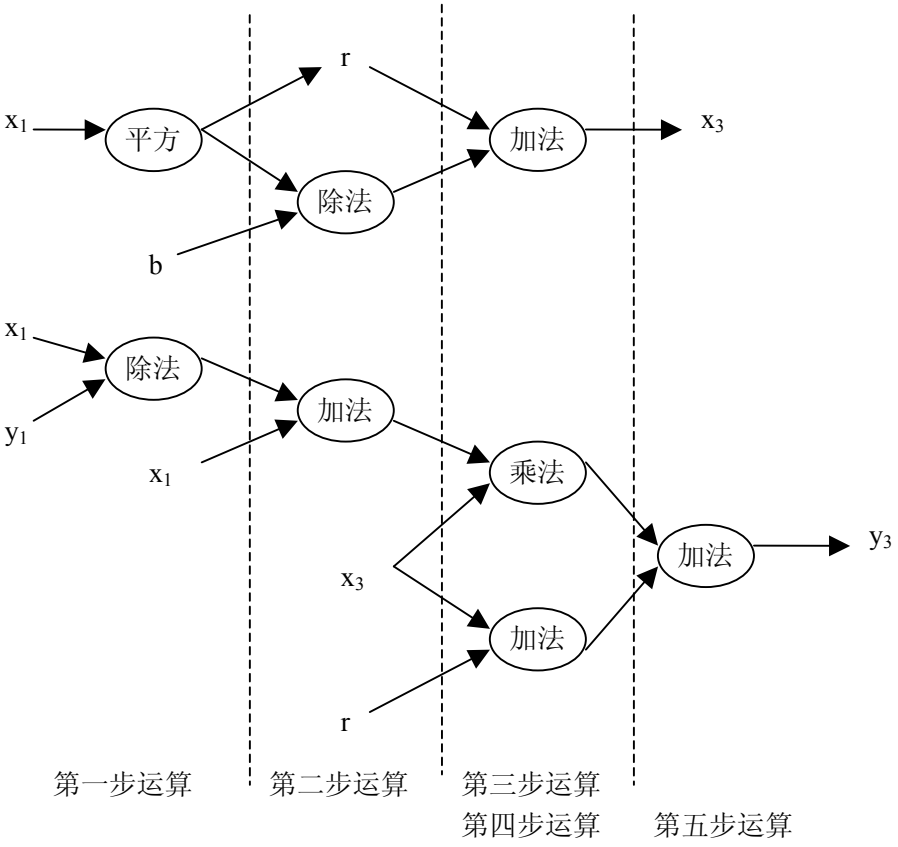

图 4.4（d） 二进制域倍点运算流程图

图 4.4（d）是二进制域倍点运算流程图。对于 x_3 的运算只需要 3 步：平方→除法→加法。计算 y_3 需要再增加两步运算：乘法→加法。二进制域倍点运算只需要 5 个计算步骤：平方→除法→加法→乘法→加法。这个运算看似运算步骤比较少，只要 5 步，但是它所耗费的资源并不比前面的运算少。因为在做第一步平方运算时同时还要做一个除法运算。在运算过程中并不需要增加寄存器来存储中间结果。

通过图 4.4 的四幅图就可以很好的回答在本节刚开始提出的三个问题。从图上可以很清楚的看出素数域点加和倍点运算以及二进制域的点加和倍点运算这四种运算如何去进行，它们分别需要几个步骤，而每个步骤所执行的运算是哪些。从图中可以发现，对于有限域的乘法和除法以及平方这三种复杂运算，在这四种运算的各个步骤中最多都只执行一次，因此在下一级模块中只设计一个乘法器，一个除法器和一个平方器最为理想。有限域的加法运算和减法运算在某些步骤中

会运算两次，因此为了提高运算速度可以设计两个加法器和两个减法器。对于这一级模块的灵活设计是我们整个设计的一个优点。正是由于在本设计中把点乘划分为三级模块，每一级模块单独设计，互相没有影响，才使得整个设计灵活多变，具有很强的实用性。

4.4 有限域运算模块的硬件结构

有限域运算模块是点乘运算的最后一级模块，也是设计难度最大的一级模块。前两级模块的设计重点主要是在控制电路上，而这一级模块将会承担起点乘运算中所涉及到的所有有限域的运算。它的设计难点在于运算单元的设计。由于运算单元的运算性能将直接影响整个点乘的运算性能，因此有限域运算模块也是最关键的一个模块。作为一个运算单元的硬件设计来说，首先考虑的就是算法的选择。一个适合于硬件实现的算法对于运算单元的硬件设计至关重要。在本文的第三章主要是针对算法的研究，最后提出了有限域混合算法（算法 46）。混合算法是一个可以实现有限域加法、减法、乘法、乘方以及除法等所有有限域运算的算法，并且可以同时支持素数域和二进制域两个有限域，它将是这一模块设计的主要依据。

4.4.1 有限域求逆和除法运算的电路结构

下面我们具体的分析一下算法 46 所需要的硬件资源，并且给出这一级模块的电路结构图。算法 46 的创新点之一是增加了“mode”控制信号，来判断具体执行哪种有限域运算。在算法 46 中一共有 9 种运算模式，涵盖了素数域和二进制域上的所有运算。

这级模块的设计是以有限域求逆电路为基础进行改进的，因此先来分析求逆算法所用到的硬件资源。图 4.5 给出了一个简单的结构图，从图上可以看到主要分为三个部分：

1、寄存器堆主要是由 4 个寄存器组成，它们分别是 u、v、x_1、x_2。这四个寄存器必不可少，用来寄存求逆或除法运算的中间结果。由于有限域求逆或除法运算一般需要 2m（m 为输入数据的二进制位长）个时钟周期，而在进行运算的 2m 个时钟周期中 u、v、x_1、x_2 这四个寄存器里的数据会不断变化，因此是计算

的关键。但是当利用算法 46 来计算有限域上的其它运算时并不需要这四个寄存器，为了提高硬件资源的利用率，在本设计中把这四个寄存器也尽可能的利用到其它的运算中去，从而不再引入新的寄存器，节省了硬件资源。整个有限域运算模块中寄存器的数量只有 4 个，即 u、v、x_1、x_2。

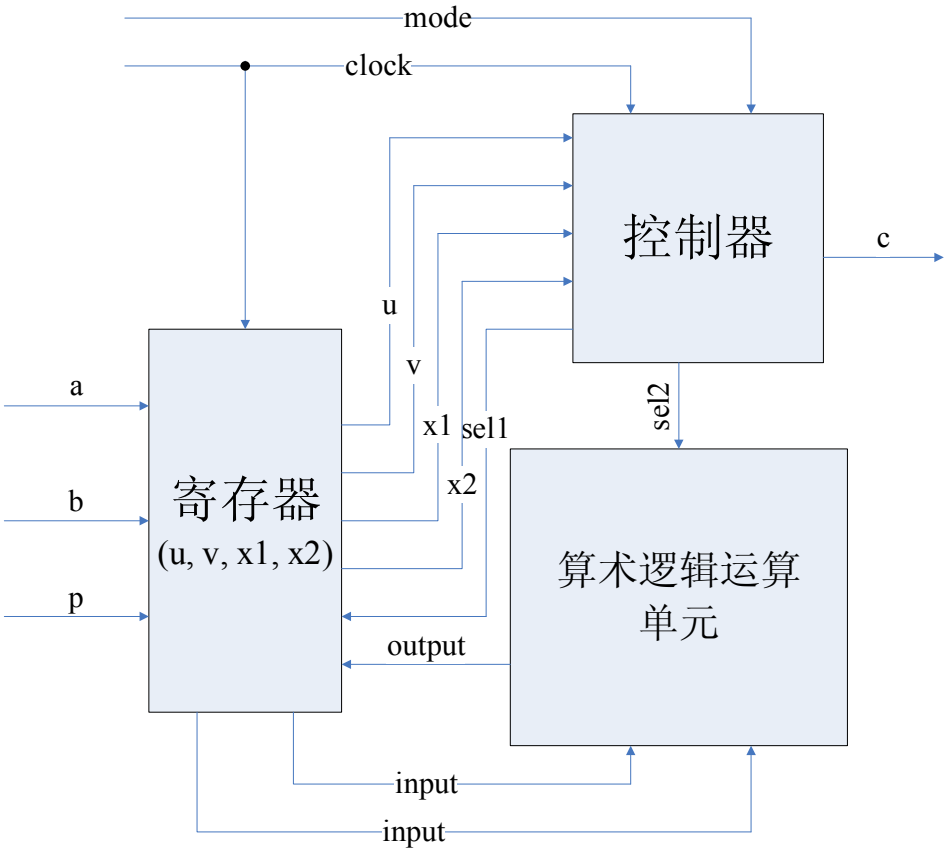

图 4.5　有限域求逆和除法运算电路结构示意图

2、为了提高有限域求逆和除法运算的计算效率，本设计对于运算的控制条件作了改进。这一点在上一章介绍算法的过程中已经作了详细的说明，这里不再赘述。不过要想很好的用硬件来实现这些条件判断，要对这些条件语句进行详细的分析。算法 46 的步骤 6 中出现的条件判断主要有三种：对寄存器中数值的直接判断，例如$u \neq 1$等；对寄存器数值的奇偶性的判断；还有两个寄存器数值的比较。

前两种判断用硬件实现起来很容易，只需要对寄存器中某些位的数值进行判

断即可，并不消耗很多硬件资源。但是对于$u \geq v$或$x_1 > x_2$这样的数值比较，要重点设计。因为这些判断对于硬件实现来说需要增加比较器或减法器这样的电路资源，更重要的是由于 u、v、x_1、x_2这四个寄存器的位数较长，因此可能花费在比较上的时间过长，增加关键路径的长度，使得运算速度受到影响。因此这个问题必须解决。

在本设计采用的解决办法是先预计算，再判断计算结果的符号位的办法。在算法的第 6.3.3 步涉及到了 u、v 以及 x_1、x_2这两组寄存器中数值大小的判断，而在条件判断之后，都会出现求 u 和 v 的差和 x_1和 x_2的差的运算。正是看到了这个特点，在具体的硬件设计中我们把后面的求差计算放到条件前面来计算，即先求差，再判断。由于在计算之前不能确定哪个寄存器的数值较大，因此计算结果有可能出现正数和负数两种可能性。通过对计算结果的符号位的判断，即可明确结果的正负性。如果结果为正，则继续做后面的运算，如果结果为负，则需要将被减数和减数位置颠倒后重新计算。这种改进避免了用作条件判断的比较器，取而代之的是对符号位的判断，大大简化了控制电路的设计，同时也减少了关键路径的计算时间，为整个电路时钟频率的提高奠定了基础。但是当预计算的结果为负数时，需要多用一个时钟周期来重新计算两个寄存器的差。用一个时钟周期来换取控制电路的简化和较短的关键路径是完全值得的。这一重要改进当然也用到了有限域运算模块的设计当中。

3、有限域求逆和除法运算电路结构的最后一块电路是算术逻辑运算单元。这一部分电路完全是组合电路，用来完成有限域求逆和除法运算过程中所需要的计算。主要包括一个加法器、一个减法器和一个按位异或运算单元。由于这三个电路结构是按照标准设计单元来设计的，并没有做任何的改动，因此在这里就不详细的介绍它们各自的电路结构了。

4.4.2 有限域其它运算的电路结构

把求逆和除法运算的硬件结构介绍清楚以后，再来看其它的运算。在算法 46 前面的 5 个步骤是用来完成有限域上的其它运算的。这些运算相对于求逆和除法运算会简单一些，而我们的设计思想就是在有限域求逆和除法运算电路结构的基础之上进行改进，使其具备完成其它运算的功能。

算法 46 中的步骤 1 和步骤 2 完成素数域上的加法和减法。这两步运算所需要的电路资源就是一个加法器和一个减法器，可以复用求逆和除法运算电路的中的运算单元。需要特别说明的是在步骤 1.2 中有$c \geq p$这样一个条件判断，我们采用上面的改进，先进行预计算再判断结果的方法，使得我们的硬件结构更有效率。步骤 3 用来完成二进制域的加减法运算，同样复用按位异或运算单元即可。因此可以看出，只要对有限域求逆和除法电路结构中的控制单元进行简单的改进就可以使其完成有限域上的加法和减法运算。

算法 46 的步骤 4 主要是完成素数域上的乘法和平方运算。这个运算是我们改进以后的算法，非常适合于硬件实现。考虑运算所需要的时钟周期，乘法运算是素数域上的求逆和除法运算所需时钟周期数相同，对于 m 位的乘法或乘方运算需要 2m 个时钟周期。但是这个运算的控制电路和求逆和除法运算相比非常简单。主要的运算资源仍然是一个加法器和一个减法器，完全可以复用求逆和除法运算的运算单元。对于这部分的设计还要在控制单元进行相应的调整即可。和步骤 4 非常类似的步骤 5 完成二进制域上的乘法或平方运算。按位异或运算单元是它主要的资源，同样可以复用。但是二进制域乘法或平方运算只需要 m 个时钟周期即可完成二进制域上 m 位的乘法或乘方运算。

通过前面的分析可以看出，把有限域求逆和除法运算单元改进成可以支持其它有限域运算的电路主要的工作就是要重新设计控制单元。而对于寄存器堆和算术逻辑运算单元这两部分电路来说并不需要重新设计。在增加了有限域上其它运算功能之后并没有增加寄存器堆和算术逻辑运算单元的硬件资源，这一点也保证了最后的运算模块与求逆和除法运算电路相比只增加了很少的面积，增加的这部分面积全部来自控制单元的重新设计。

在整个模块的控制单元的设计上我们采用有限状态机的形式。具体的结构框图如图 4.6 所示。在图 4.6 中只是表示出了最为重要的状态以及相应的控制信号，但控制单元的大体结构已经可以很清楚的看出来。在不做计算时控制单元处于“空闲”状态，当计算数据到来时会立即触发“开始”状态，进行数据的初始化工作。然后根据输入信号“mode”选择一种有限域上的“运算”状态，当计算结束后会将结果信号“result”置“1”，表示计算结束，最后是“结束”状态，将计算结构存放到输出寄存器中。

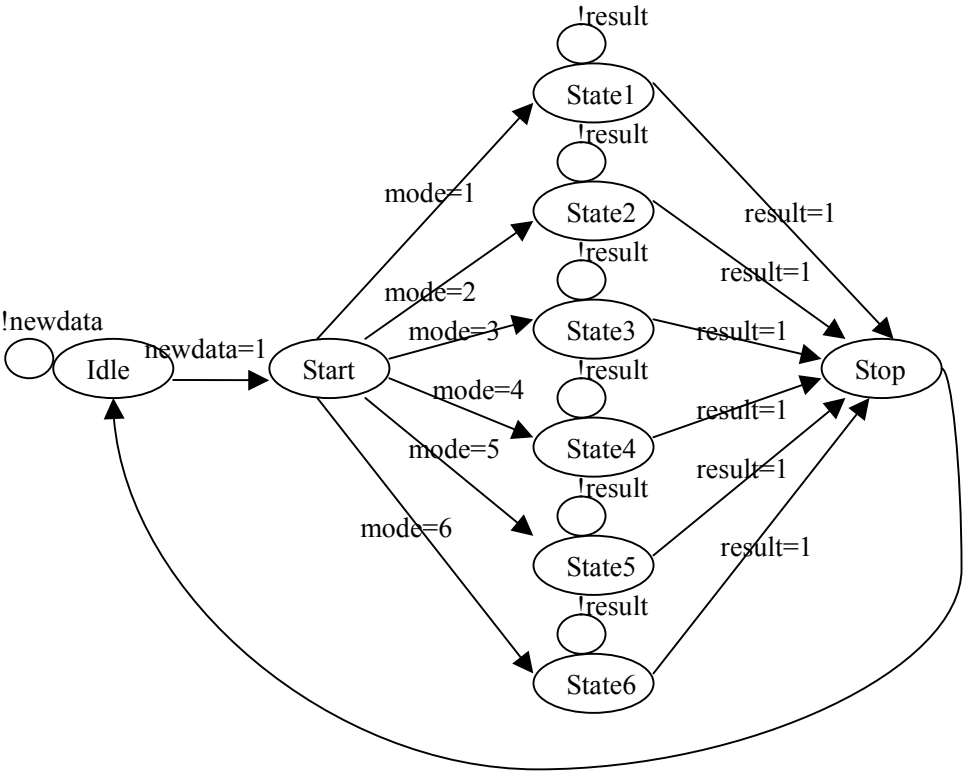

图 4.6　有限域运算模块控制单元有限状态机简图

4.5 本章小结

在点乘模块的电路设计过程中我们主要做出了三点创新：

1、点加和倍点模块的电路结构的分析与改进。在前面的阐述中可以看出我们对素数域和二进制域上的点加和倍点运算作了非常深入的分析，提出了自己对这部分电路的独到理解。

2、求逆和除法电路中比较运算电路结构的改进。在求逆和除法电路中出现了一个比较运算。如果直接比较两个 m 位的数据，将要花费大量的时间，同时还会占用硬件资源。我们对这个控制运算做出了非常重要的改进，采用先预计算后判断的方法。这一点改进缩短了整个电路的关键路径。

3、有限域运算模块控制电路的改进。有限域运算模块非常复杂，需要整合双域的所有运算，是整个 ECC 点乘电路的运算核心。必须要设计出非常高效的控制电路，保证各个运算模块高效进行。

在提出以上三点创新的同时，我们也完成了点乘运算模块的电路设计。这是本设计的又一个非常重要的阶段性成果。算法的创新以及电路结构的创新，都是在参考了大量的学术文献基础上做出来的。从最后对电路的仿真和测试结果来看我们创新是具有现实意义的，这种改进通过数据可以很好的体现的。我们将在下一章讲述电路的仿真和测试结果。

第五章　功能仿真与电路测试结果

在完成了算法的改进和电路结构设计这两个非常重要的工作之后，我们对所设计的电路进行了功能仿真，并且对于电路的 VLSI 实现和 FPGA 实现都做了仿真，也得到了很多重要的数据来验证我们的设计的正确性。随后我们还进行了 FPGA 电路测试，所得到的电路参数基本上达到了设计的预期。

在 5.1 节将介绍电路的 VLSI 设计和功能仿真结果。5.2 节讲述 FPGA 设计的具体工作。5.3 节将对本设计的电路测试结果进行分析与比较，综合的评定整个设计工作的效果。最后在 5.4 节是本章小结。

5.1 VLSI 设计

通过前面的介绍可以知道，本设计的重点创新部分体现在了算法上的创新以及电路结构的改进。而这些创新是否成功要用最后的电路实现方案所得到的仿真数据和测试数据来检验。本节将详细介绍 VLSI 实现方案。

5.1.1 VLSI 设计流程

在 VLSI 设计中，我们将这些创新通过硬件描述语言 verilog-HDL 编写成代码，在 Sun 工作站的平台上通过 Synopsys 的 Design Compiler 进行了综合设计，所采用的工艺为中芯国际的 0.18 um CMOS 工艺。并且对最后的综合结果进行了综合后仿真。具体的设计流程如图 5.1 所示。

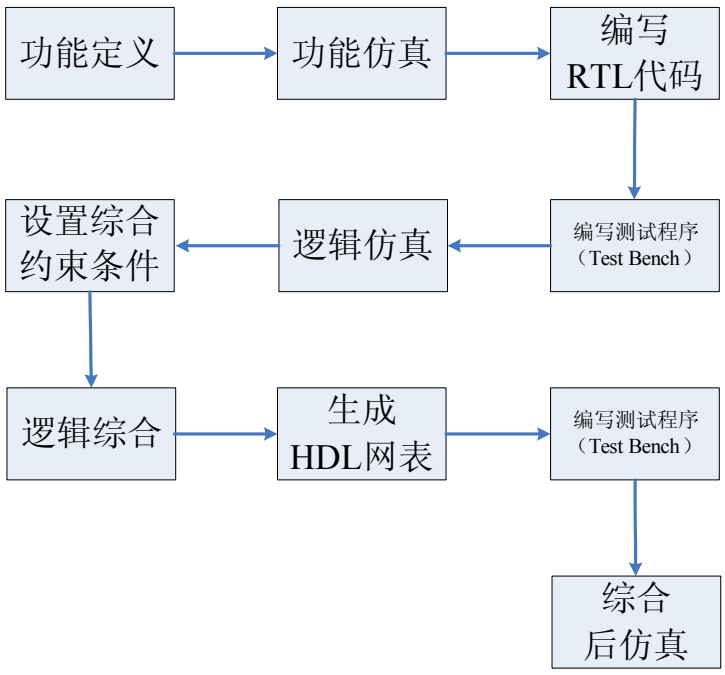

图 5.1　VLSI 设计流程

5.1.2 VLSI 功能仿真结果

在上一章讲述硬件结构的时候我们已经将点乘运算电路分解成为三个主要模块，在仿真的过程中我们首先对三个模块分别仿真，然后再将三个模块合在一起，对整个点乘电路进行了仿真。

表 5.1　素数域仿真结果（模数 p = 19）

运算模式 （mode）	输入数据 （a）	输入数据 （b）	输出结果 （c）
0（加法）	10	15	$10+15=6 \bmod 19$
1（减法）	10	15	$10-15=14 \bmod 19$
2（乘法）	10	15	$10\times15=17 \bmod 19$
3（除法）	10	15	$10\div15=7 \bmod 19$
4（求逆）	10	N/A	$1\div15=14 \bmod 19$

表 5.2　二进制域仿真结果（取模多项式 f(x) = 10011）

运算模式（mode）	输入数据（a）	输入数据（b）	输出结果（c）
5（加法）	1111	1010	0101
6（减法）	1111	1010	0101
7（乘法）	1111	1010	1100
8（除法）	1111	1010	1111
9（求逆）	1111	N/A	1000

表 5.1 和表 5.2 给出了本设计在对有限域运算模块仿真时所采用的一组数据。由于本设计的一大创新点就是输入数据位数的长短并不受寄存器的限制，也就是本设计的控制单元可以根据输入数据位数的长短自动调节计算次数。为了充分验证这一创新点，我们选用的表 5.1 和表 5.2 的数据分别在 8 位寄存器、16 位寄存器、32 位寄存器、64 位寄存器、128 位寄存器、256 位寄存器和 512 位寄存器这些不同位长的寄存器所对应的电路中进行仿真，仿真结果完全正确，达到了设计之初的效果。同时对于位数较长的寄存器例如 128 位寄存器、256 位寄存器和 512 位寄存器，我们还选用了一些官方文献中 [1]提到过的一些标准数据进行了功能测试，最后的结果都是完全正确的。

图 5.2 给出了有限域运算单元做素数域除法运算的仿真波形图，这个波形图是本设计综合后的网表用 Candence 的波形软件显示出的结果。从图上可以看出，运算结果完全正确，并且在每一个时钟周期里，各个寄存器中的数据也是准确无误。图 5.2 中的仿真波形是在 8 位寄存器对应的电路中进行仿真的。由于输入数据都是四位的数据，因此只需要 8 个时钟周期就可以完成运算，并且输出结果。从图 5.2 中的“clk”信号就可以看出从写入数据到寄存器到最后输出计算结果一共用了 8 个时钟周期，与理论设计的结果完全一致。这一点也验证出计算所需要的时钟周期与输入数据的位长有关，而与电路寄存器的位长无关。我们的验证过程使用了大量的数据，也得到了很多的波形，由于篇幅有限，这里只给出了素数域上难度最大的除法运算波形图。

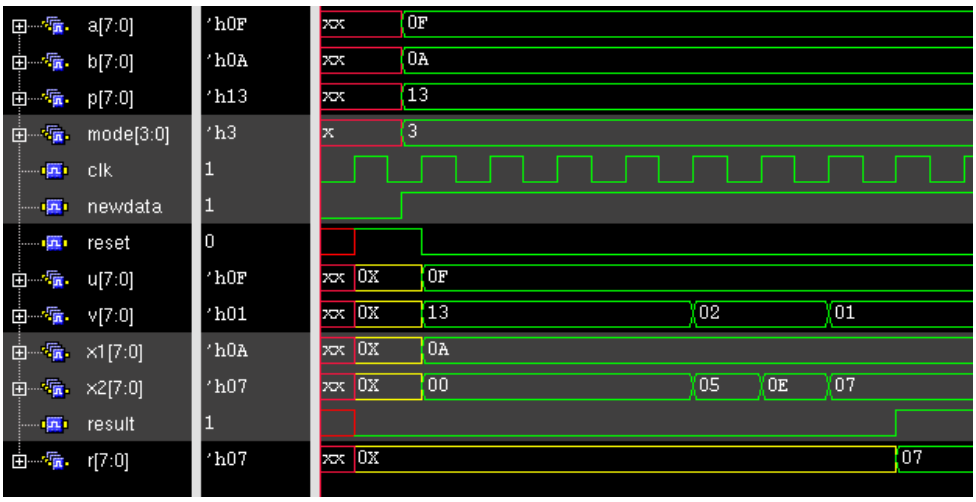

图 5.2　素数域除法运算的仿真波形图（$10 \div 15 = 7 \bmod 19$）

有限域运算模块的功能仿真结果完全正确之后，我们开始对整个点乘运算单元进行功能仿真。表 5.3 给出了我们做点乘运算所选用的输入数据，素数域选用的椭圆曲线是 $F_{29}: y^2 = x^3 + 4x + 20$，二进制域选用的椭圆曲线是 $F_{2^4}: y^2 + xy = x^3 + 8x^2 + 9$。在两种有限域上各选择了两组不同的 k 值，为了提高仿真速度，在 8 位的寄存器环境下进行点乘运算，素数域和二进制域上的点乘运算结果都是完全正确的，这充分说明我们对于椭圆曲线加密算法的改进是成功的。（关于点乘运算测试数据的选择及其可靠性，请参见本文附录）

表 5.3　点乘测试数据

椭圆曲线 E	P	k	kP
$F_{29}: y^2 = x^3 + 4x + 20$	(1,5)	00001111	(3,1)
$F_{29}: y^2 = x^3 + 4x + 20$	(1,5)	00100000	(6,17)
$F_{2^4}: y^2 + xy = x^3 + 8x^2 + 9$	(1000,0001)	00000011	(1100,0000)
$F_{2^4}: y^2 + xy = x^3 + 8x^2 + 9$	(1000,0001)	00001000	(1100,1100)

在功能仿真结果完全正确的基础上我们又作了另一个重要的工作：在同样的

仿真环境下对求逆运算电路和混合运算电路进行综合。这个工作对于验证我们对电路结构的改进效率很重要。我们所设计的有限域运算模块是在求逆运算电路的基础上改进得来的，是在求逆运算的基础上加入了加法、减法和乘法运算。从理论上说，由于增加了多种运算功能，且控制单元更加复杂，因此组合面积一定会增加。但是由于有限域运算模块和求逆运算电路都只有 4 个寄存器，因此时序面积不会有明显的变化。这种改进效果如何，我们将通过具体的仿真数据来检验。

表 5.4 给出了我们验证的结果。求逆运算电路和混合运算电路（有限域运算模块）综合的环境是完全一致的，都是在时钟频率 200MHz 的环境进行综合，并且两个电路我们都采用了 256 位寄存器的设计。

表 5.4　综合结果比较

	时钟频率	寄存器位数	组合电路面积	时序电路面积	电路总面积
混合运算电路	200MHz	256 位	44623（门）	10005（门）	54628（门）
求逆运算电路	200MHz	256 位	30758（门）	9907（门）	40665（门）

在表 5.4 的后三栏给出了两个电路的组合面积、时序面积和总面积。虽然这只是一个综合结果，并不能说明电路的实际面积大小，但是用作数据的比较还是能说明问题。通过对比可以看出，混合运算电路由于要增加很多运算，因此组合面积比求逆电路增加了 45%。但是，由于我们对于控制单元设计的很巧妙，使得混合电路的时序部分和求逆电路相比，几乎没有任何面积的增加。最后，混合运算单路的总面积仅比求逆运算电路总面积多了 34%。在求逆运算电路的基础上，增加了加法、减法和乘法运算，而电路面积仅增加了 34%，证明这是一个效率很高的改进。

5.2 FPGA 设计

本设计在电路结构上面并没有针对某一型号的 FPGA 芯片做专门的设计，我

们所设计的电路是在所有 FPGA 中通用的。

5.2.1 FPGA 设计流程

FPGA 设计的工作内容很简单。主要流程如图 5.3 所示，在本设计中我们选用了 Xilinx 公司生产的 Virtex2 系列的 XC2V3000 芯片，封装标准为 fg676。以我们前面已经设计好的 ECC 点乘运算 verilog-HDL 代码作为输入文件，通过专业软件来完成 FPGA 硬件电路的设计。最后硬件测试的结果是完全正确的，这再一次验证了我们的算法设计和电路结构设计是完全成功的。

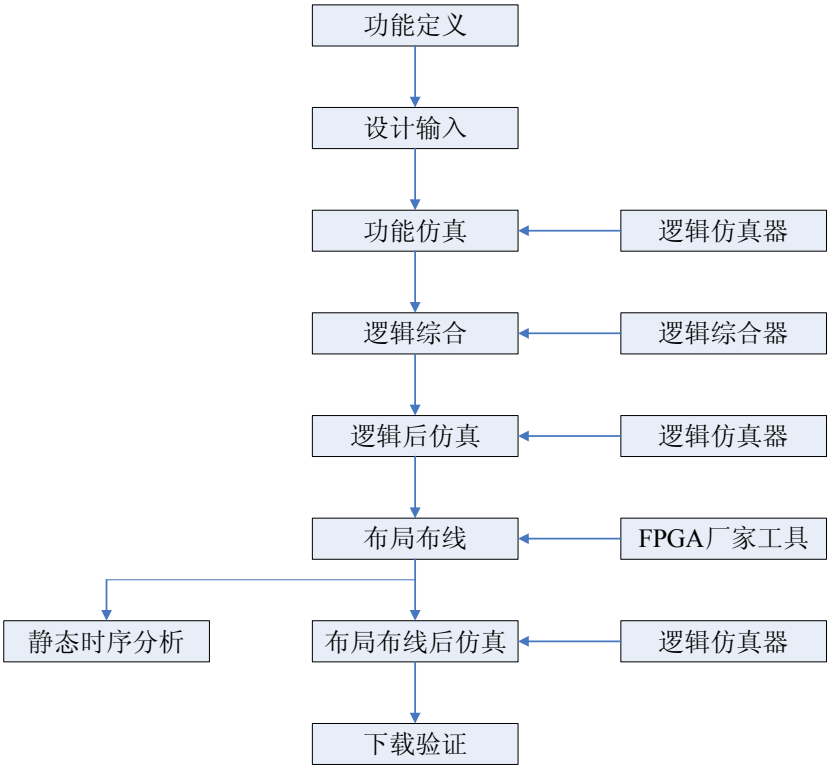

图 5.3　FPGA 设计流程

5.2.2 FPGA 电路测试结果

FPGA 设计的输入文件仍然是前面已经仿真过的 verilog-HDL 代码，其仿真结果在前面一节已经详细介绍过了，因此这里重点介绍 FPGA 的硬件参数，具体

的 FPGA 设计平台如图 5.4 所示。我们对点乘电路的三级模块分别进行了测试，并且选择不同位长的寄存器对 ECC 运算电路进行了详细的测试。这部分的工作量是非常大的，不过我们取得了很重要的测试结果。

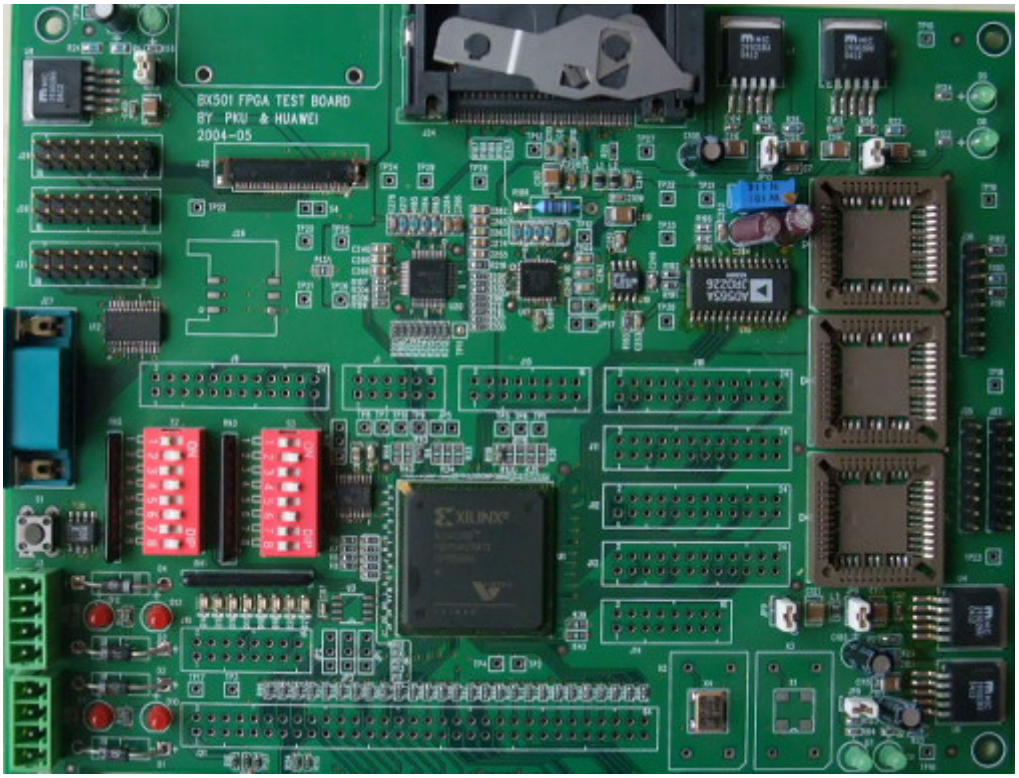

图 5.4　FPGA 设计开发平台

表 5.5　有限域运算模块 FPGA 设计的面积结果

寄存器位长	Slices	Flip Flops	4 input LUTs
32 位	1523	249	2843
64 位	2768	466	5154
128 位	5234	835	9763
256 位	13048	1786	24009

表 5.6　有限域运算模块 FPGA 设计的频率和功耗结果

寄存器位长	最高频率	电路总功耗（3.3v）
32 位	73.942 MHz	665 mW
64 位	62.223 MHz	702 mW
128 位	59.917 MHz	772mW
256 位	40.972 MHz	864 mW

表 5.5 和表 5.6 都是针对点乘电路的第三级模块有限域运算模块进行的电路测试结果。在前面的章节提到点乘电路的电路结构分为三级：点乘模块、点加倍点模块和有限域运算模块。前面两级模块只是一些控制电路，最后一级模块是整个电路的运算模块，也是我们重点设计的模块，因此我们对其进行了专门的设计和测试。

从硬件参数来看，电路所消耗的硬件资源基本上和数据的位长成线形增长，当数据的位数扩大一倍时，消耗的资源也基本上扩大一倍，这也是 ECC 点乘运算单元的普遍规律，我们的设计满足这一规律。只是在寄存器的位数增加到 256 位时面积稍大，这主要是由于加法器等运算单元复杂度上升造成的。在 XC2V3000 这款芯片上 256 位的运算电路时钟频率最低为 40.972 MHz，表现还是不错的。功耗的表现也令人满意。

表 5.7　点乘电路 FPGA 设计的面积结果

寄存器位长	Slices	Flip Flops	4 input LUTs
32 位	1940	433	3628
64 位	3405	788	6366
128 位	6485	1423	12046

表 5.8　点乘电路 FPGA 设计的频率和功耗结果

寄存器位长	最高频率	电路总功耗（3.3v）
32 位	71.426 MHz	685 mW
64 位	62.707 MHz	723mW
128 位	54.058MHz	791mW

表 5.7 和表 5.8 给出了点乘电路三级模块合在一起的整体电路测试结果。把这两个表格中的数据和表 5.5 和表 5.6 中的数据作比较，就可以看出问题所在。从电路结构上来说，前面两级模块只是一些控制电路，如果单独测试，它们占用资源非常少。主要的运算都集中在第三级模块，这级模块负责所有有限域上的运算，它应该占有绝大部分的资源。不过通过表 5.5 和表 5.7 的对比可以看出，加上前面两级模块以后，电路的总面积大约为原来的 1.5 倍。这个数据超出了我们的预测，通过对电路所使用的资源可以看出问题出现在了第三级模块的设计。我们把第三级模块设计成了可以完成有限域所有运算的电路模块，对于这个模块本身来讲节省了大量的面积，但是由于只有一个运算模块，每次只能输入一组数据进行计算，这将在很大程度上增加前面两级模块控制电路的优化难度。所以在前面两级模块的设计中要加入寄存器存储中间结果，同时由于控制电路的路径过长，需要插入大量的缓冲器，也影响到了最终电路的速度。

5.3 电路测试结果分析与比较

目前我们已经完成了椭圆曲线加密算法的核心运算点乘的算法设计，点乘模块的电路结构设计以及点乘模块 FPGA 方案的电路设计。在已经完成的工作中得到了很多重要数据，我们把这些数据总结下来，和国内外发表的一些文献进行了比较。具体的内容将在本节做出详细介绍。

对于我们这样一个设计来说，计算速度是非常重要的，对于算法的诸多创新就是为了提高计算能力。所以我们主要比较了各个设计的计算速度。

首先来看一下本设计的计算速度。在素数域 F_p 上，用域运算表示的点乘运行时间为：

$$2.5mS + 3mM + 1.5mI$$

其中 S 表示平方运算时间，M 表示乘法运算时间，I 表示求逆运算时间。这个式子中并没有表示出加减法的运算时间，因为它们和平方、乘法以及求逆运算比较起来可以忽略。但是通过我们对于第二级模块的改进，将乘法和求逆运算统一成除法运算，缩短了计算时间。如果把加减法的运算时间也考虑进去，用 A 来表示其运算时间，最终点乘的素数域 F_p 运算时间可以表示成：

$$2.5mS+1.5mM+1.5mI+11mA$$

在二进制域 F_{2^m} 上，点乘的标准运算时间可表示为：

$$1.5mS+3mM+1.5mI$$

通过我们的改进，如果考虑加减法的运算时间，最终点乘的计算时间为：

$$1.5mS+1.5mM+1.5mI+8mA$$

在本设计中，有限域上的所有运算都是由有限域运算模块来完成，对于每一种运算，都有固定的时间。综合起来，本设计在素数域 F_p 上完成 m 位的点乘运算所需时钟周期数为：

$$11m^2+22m$$

在二进制域上完成 m 位的点乘运算所需时钟周期数为：

$$6m^2+8m$$。

表 5.9　FPGA 设计速度比较

	有限域	FPGA 类型	时钟频率	计算时间
本设计	GF(p) p 为 163 位 和 GF(2^{163})	FPGA Xilinx Virtex-II XC2V3000	70MHz	素数域 4.23ms； 二进制域 2.29ms
文献[97]	GF(p) P 为 192 位	FPGA Xilinx Virtex-II	66MHz	4.8ms
文献[98]	GF(p) P 为 163 位	Xilinx V1000E	91.3MHz	14.4ms
文献[99]	GF(2^{163})	FPGA Altera EP1M350F780C5	N/A	0.9368ms
文献[100]	GF(2^{163})	FPGA Xilinx Virtex-II	N/A	1.9ms

表 5.9 比较了我们的设计和国内外一些比较好的设计。受到选用的 FPGA 芯

片的限制，本设计在时钟频率上比较慢，因此影响了 ECC 的计算时间。但是本设计的算法上的优势还是很明显，计算次数少，而且可以支持素数域和二进制域两种有限域，功能强大。

表 5.10　FPGA 设计面积比较

	有限域	FPGA 类型	面积参数
本设计	GF(p)p 为 163 位和 GF(2^{163})	FPGA Xilinx Virtex-II XC2V3000	8247 Slices
文献[98]	GF(p) P 为 163 位	Xilinx V1000E	6 K Slices
文献[101]	GF(2^{163})	FPGA Xilinx Virtex-E XCV2600E	15280 Slices
文献[102]	GF(2^{163})	Xilinx XC4VLX80	24263 Slices

表 5.10 比较了本设计和一些设计在面积上的数据，所有的设计都是采用 FPGA 硬件设计方案。从数据中可以看出，本设计在可以完成素数域和二进制域两种有限域加密运算的基础上面积仍然是最小的。本设计在面积上的优势主要有两个原因：第一，本设计采用自行设计的算法，将两个有限域上的运算在算法层就做了很好的合并和优化，并且省去了平方运算单元和取模运算单元，因此节省了硬件资源；第二，本设计全部采用最基本的电路结构，没有为了追求高速而使用一些高速的电路结构，这也节省了电路面积。在表 5.10 中所提到的文献[101]和文献[102]是最新的关于椭圆曲线加密算法的研究成果，这两篇文献都采用了高速的电路设计，但是它们只支持二进制域上的加密运算，并不支持素数域运算，而且所耗费的硬件资源非常多。相比较，本设计在经济性上更为突出。

5.4 本章小结

本章主要介绍了椭圆曲线加密算法的核心运算——点乘的电路设计结果。对于点乘运算算法我们做了 VLSI 和 FPGA 两种实现方案：VLSI 实现方案偏重于

电路的功能仿真和对算法创新的验证；FPGA 实现方案则是在 Xilinx 公司生产的 Virtex2 系列的 XC2V3000 芯片上完成的，偏重于电路参数的测试和对电路结构创新的验证。在整个过程中选用了多组数据进行全面的仿真，确保最后数据的准确性，同时还将本设计所得到的参数和其它文献中的参数做了详细的比较和分析。

我们所设计的电路可以支持素数域和二进制域两种有限域的椭圆曲线加密运算，并且支持 256 位以下任意位长的数据加密（最高位长由电路的存储器决定）。当输入数据是 163 位，电路的时钟频率为 70MHz 时，完成素数域点乘运算仅需要 4.23ms，完成二进制域点乘运算需要 2.29ms，面积参数和同类设计相比较优势也很明显。本设计无论是在速度上还是在面积上都具有很强的竞争力。而在功能上本设计在所有发表过的文献中最为强大，支持素数域和二进制域两种有限域，并且支持任意位长的加密运算。总体来说，本设计成功的完成了椭圆曲线加密算法的运算电路的硬件设计。

第六章　总 结

这个项目从 2004 年 9 月开始启动，至今已经作了三年多的时间。在这一部分将会对整个项目做出总结。在 6.1 节将讲述这个项目研究的意义，6.2 节总结我们的研究工作，6.3 节总结创新成果，6.4 节给出后续研究工作的展望。

6.1 研究意义

本项目的主要的研究工作是：研制一款可以同时支持素数域和二进制域椭圆曲线加解密的运算芯片。当我们查阅了大量的国内外文献之后发现可以同时支持双域加密的椭圆曲线加密芯片非常少，因此这个选题是具有相当难度的。但是目前在我国信息安全领域，公钥体制加密算法多采用 RSA 算法，采用 ECC 算法的相对较少，这个项目对我国加密技术的进一步发展具有巨大的推动作用，同时一款高效、兼容性强、可移植性好的椭圆曲线加密芯片也具有十分可观的市场应用前景。因此这个项目具有非常重大的现实意义。在这种背景下，我们选择了这个项目进行研究。

6.2 研究工作

研究工作一共分成了四个阶段来进行：

第一个阶段：背景研究。在这一阶段主要的工作是查阅相关书籍和国内外的重要文献。为了做好设计，除了了解椭圆曲线加密算法本身的知识以外，还需要对密码学领域、公钥加密体制领域以及各种加密算法进行系统的掌握，因此需要阅读大量的学术文献，深入了解椭圆曲线加密算法的一系列相关知识。在这段时间，作者阅读了四本相关书籍，查阅了 400 多篇国内外关于椭圆曲线加密算法的重要文献。

第二个阶段：算法研究。在对椭圆曲线加密算法及其相关背景充分了解以及参考了国内外在该方向上的研究成果的前提下，我们确定了研究方向：研制一款可以同时支持素数域和二进制域的高效椭圆曲线加密芯片。这个方向的重点是要支持素数域和二进制域双有限域上的加密运算。

首先，对于一款可以同时支持素数域和二进制域的高效椭圆曲线加密芯片来说，并没有现成的算法，需要我们重新设计可以同时支持双有限域的算法；其次，对于数字电路的设计来说，需要我们对已有的算法进行改进，使得新算法适合于硬件实现；最后，对于数字电路设计来说，如果能够在算法设计阶段提高运算效率，简化运算步骤，是最为理想的。正是基于上述三方面的考虑，我们对于算法设计投入了大量的时间和精力，这一阶段的工作量很大，也是整个研究工作最为重要的一个阶段。

第三个阶段：电路结构研究。这个阶段的主要工作就是要把前面已经设计好的算法映射到电路上。我们根据椭圆曲线加密算法的核心运算——点乘的运算特点，把电路分为三级模块。详细分析了每级模块所包含的运算种类以及运算复杂度，并以此为依据设计了控制电路。最终完成了整个电路结构的设计。

第四个阶段：功能仿真与数据测试。这个阶段主要是用工具进行电路设计，我们按照标准的数字电路的设计流程进行。在设计过程中应用到了大量的工程软件，在编写代码和功能仿真过程中用到了 Cadance 公司的 EDA 工具，在后面的综合设计阶段则采用 Synopsys 公司的 EDA 工具。综合后仿真结果的正确充分说明我们对于算法的改进是成功的。接下来我们又针对 FPGA 的实现特点，对电路进行了必要的改进和调试，完成了 FPGA 的测试。在这个过程中，我们有针对性地选用了大量的参数，从输入数据的位数、电路的运算速度以及电路所消耗的硬件资源等不同方面对我们所设计的电路进行测试，最后的结果完全符合我们的预期目标，体现了我们对于算法以及电路结构改进的成果。通过和国内外相关设计的比较，我们的设计功能强大，在运算速度以及面积等参数上也体现了一定的优势。

6.3 创新成果

对于椭圆曲线加密算法的研究及其硬件实现这个课题来说，创新点主要来自三个方面：功能创新、算法创新和电路创新。

第一，功能创新。本文提出了一种椭圆曲线加密算法的核心运算——点乘的硬件实现方案，这个方案可以同时完成二进制域和素数域上的任意椭圆曲线的加密运算，并且对于椭圆曲线的位数没有约束。其中有限域运算模块作为一个独立

的电路模块，除了可以作为 ECC 的运算模块使用之外也可以应用到其它的电路当中。

第二，算法创新。为了实现支持双有限域运算的功能，本设计参考了两种有限域上的大量算法，并对各种算法进行分析比较，最终设计出了可以支持两种有限域的各种算法，如表 6.1。针对各个算法的创新，主要实现了以下两个目标：第一，对于算法的改进使其更适合于硬件实现；第二，对于算法的改进使其运算效率更高。

表 6.1　算法创新点总结

算法	创新点
算法 3：改进后的点乘二进制方法	1 个创新点
算法 29：同时完成乘法和约减运算的素域乘法算法 算法 30：二进制域上的乘法算法	3 个创新点
算法 45：改进后的同时支持素数域和二进制域的统一求逆算法	4 个创新点
算法 46：同时支持双有限域运算的混合算法	7 个创新点

第三，电路创新。电路创新是在算法创新取得成功之后，在功能正确的基础上进一步完成的。这部分的创新主要有三点：

1、点加和倍点模块电路结构的分析与改进。

2、求逆和除法电路中比较运算电路结构的改进。

3、有限域运算模块控制电路的改进。

6.4 研究展望

通过前面的介绍，我们已经把椭圆曲线加密算法这个项目完整地研究工作展现出来。在这个工作的基础上，我们从三个不同的方面提出研究展望：

1、在设计 ECC 点乘运算的过程中设计出了一个可以做有限域算术运算的芯片。这个运算芯片既可以作为 ECC 的运算芯片，也可以作为 RSA 等加密电路的运算芯片或者其它芯片里的运算芯片来使用。因此可以进一步完善这个运算芯片

的设计，使其功能更加强大，应用更加广泛。

2、本设计已经完成了 ECC 点乘运算芯片的设计，可以根据国际上的标准协议进一步完成外围的设计。

3、从最后 FPGA 的测试结果来看，本设计的电路参数，如电路的时钟频率和电路的面积等，显得很平均，具有综合优势。这虽然对于普通的应用来讲，不存在任何问题，但是现在随着加密算法的应用越来越广泛，很多应用场合对于加密电路的要求也越来越高。有些地方需要高速的加密电路，有些地方需要低功耗的加密电路，有些地方对加密电路的面积要求很高。接下来的工作可以从这方面入手，设计出适合于不同应用环境的加密电路。

附录　点乘测试数据

通过前面的介绍我们知道点乘运算是一个非常复杂的运算，其运算步骤非常多，而且一般认为只有当加密数据的位数达到 163 位以上的椭圆曲线加密运算才足够安全，因此点乘运算在通常情况下都要完成 163 位以上数据的运算。如果用这么大的数据进行测试，占用的时间是非常多的。

在本设计的数据测试过程中，我们跟据本设计的特点，重点选择了 8 位以下的数据进行主要的功能测试。因为本设计的一个重要改进是对于输入数据是没有限制的，可以对输入的不同有限域上的各种椭圆曲线进行加密运算，椭圆曲线的各个参数都是可以从外部自由输入的。因此选择 8 位以下的数据进行功能测试完全可以验证整个电路的功能，并且提高了验证效率。选择 8 位以下的数据进行功能测试也便于我们监控运算的过程中每一步骤的中间结果。在 163 位的点乘测试中则只能验证最后的输出结果。

在素数域点乘运算的测试中，我们选择了素数域 F_{29} 上的椭圆曲线 $E: y^2 = x^3 + 4x + 20$，该椭圆曲线的输入数据为 $(p = 29, a = 4, b = 20)$。这个椭圆曲线的阶 $\#E(F_{29}) = 37$，因为 37 是素数，因此这 37 个元素构成有限循环群，满足椭圆曲线加密算法的要求，可以进行点乘运算。我们可以把点 $P(1,5)$ 作为生成元，通过点乘运算生成群 $E(F_{29})$ 的全部 37 个元素：

$0P = \infty$	$8P = (8,10)$	$16P = (0,22)$	$24P = (16,2)$	$32P = (6,17)$
$1P = (1,5)$	$9P = (14,23)$	$17P = (27,2)$	$25P = (19,16)$	$33P = (15,2)$
$2P = (4,19)$	$10P = (13,23)$	$18P = (2,23)$	$26P = (10,4)$	$34P = (20,26)$
$3P = (20,3)$	$11P = (10,25)$	$19P = (2,6)$	$27P = (13,6)$	$35P = (4,10)$
$4P = (15,27)$	$12P = (9,13)$	$20P = (27,27)$	$28P = (14,6)$	$36P = (1,24)$
$5P = (6,12)$	$13P = (16,27)$	$21P = (0,7)$	$29P = (8,19)$	

$6P=(17,19)$ $14P=(5,22)$ $22P=(3,28)$ $30P=(24,7)$

$7P=(24,22)$ $15P=(3,1)$ $23P=(5,7)$ $31P=(17,10)$

在二进制域上选择了二进制域F_{2^4}上的椭圆曲线$E:y^2+xy=x^3+8x^2+9$，二进制域F_{2^4}上的取模多项式为$f(x)=x^4+x+1$，或简写为$f(x)=10011$（后一种表示法只给出了多项式各项系数）。椭圆曲线$E:y^2+xy=x^3+8x^2+9$的输入数据是$(f(x)=10011,a=1000,b=1001)$。因为阶$\#E(F_{2^4})=22$，且22没有重因子，所以$E(F_{2^4})$里的元素构成有限循环群，同样满足椭圆曲线加密算法的要求，可以进行点乘运算。在椭圆曲线$E:y^2+xy=x^3+8x^2+9$上选择点$P=(1000,0001)$作为生成元，阶为11，通过点乘运算同样可以得到全部11个元素：

$0P=\infty$ $3P=(1100,0000)$ $6P=(1011,1001)$ $9P=(1001,0110)$

$1P=(1000,0001)$ $4P=(1111,1011)$ $7P=(1111,0100)$ $10P=(1000,1001)$

$2P=(1001,1111)$ $5P=(1011,0010)$ $8P=(1100,1100)$

在我们的设计中就是选用了上述数据对所设计的电路进行点乘测试的。

参考文献

[1] Darrel Hankerson, Alfred Menezes, Scott Vanstone 原著，张焕国等译. Guide to Elliptic Curve Cryptography（椭圆曲线加密算法导论）. 北京：电子工业出版社，2005 年 8 月.

[2] V. Miller, “Uses of elliptic curves in cryptography,” *Proc. Advances in Cryptology (CRYPTO ’85)*, pp. 417–426, 1986.

[3] N. Koblitz, “Elliptic curve cryptosystems,” *Math. Comput.*, vol. 48, no. 177, pp. 203–209, Jan. 1987.

[4] N. Koblitz, A. Menezes, and S. Vanstone, “The state of elliptic curve cryptography,” *Designs, Codes Cryptography*, vol. 19, pp. 173–193, 2000.

[5] Jarvinen K, Tommiska M, Skytta J, “A scalable architecture for elliptic curve point multiplication,” *Field-Programmable Technology, 2004, Proceedings*, pp. 303 – 306, 2004.

[6] Moon S, “Elliptic curve scalar point multiplication using radix-4 Booth's algorithm [cryptosystems],” *ISCIT 2004, IEEE International Symposium*, vol. 1, pp. 80 – 83,Oct. 2004.

[7] Jun-Hong Chen, Ming-Der Shieh, Chien-Ming Wu, “Concurrent algorithm for high-speed point multiplication in elliptic curve cryptography,” *ISCAS 2005, IEEE International Symposium*, vol. 5, pp. 5254 – 5257, May 2005.

[8] BRUNNER H, CURIGER A, HOFSTETER M, “On computing multiplicative inverses in GF(2^m),” *IEEE Trans Computers*, vol. 42, no. 8, pp. 1010 – 1015, 1993.

[9] MCLVOR C, MCLOONE M, MCCANNY J V, “Improved Montgomery modular inverse algorithm,” *Electronics Letters*, vol. 40, no. 8, pp. 1110 – 1112, 2004.

[10] 王健，蒋安平，盛世敏. 同时支持两种有限域的模逆算法及其硬件实现. 北京大学学报（自然科学版），第 43 卷，第 1 期，138–143. 2007 年 1 月.

[11] Chang-Soo Ha, Joo-Hong Kim, Byeong-Yoon Choi, et al, “GF(2^{191}) Elliptic

Curve Processor using Montgomery Ladder and High Speed Finite Field Arithmetic Unit," *TENCON 2005, 2005 IEEE Region 10*, pp. 1– 4, Nov. 2004.

[12] Choi Hyun Min, Hong Chun Pyo, Kim Chang Hoon, "High Performance Elliptic Curve Cryptographic Processor Over GF(2^{163})," *DELTA 2008. 4th IEEE International Symposium*， pp. 290 – 295, Jan. 2008.

[13] Hai Yan, Zhijie Jerry Shi, "Studying Software Implementations of Elliptic Curve Cryptography," *Information Technology: New Generations, 2006. ITNG 2006.*, pp. 78 – 83, April 2006.

[14] Ors S.B., Batina L., Preneel B., et al, "Hardware Implementation of an Elliptic Curve Processorover GF(p)," *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International*, pp. 433–443, June 2003.

[15] Mcivor C.J., Mcloone M., Mccanny J.V., "Hardware Elliptic Curve Cryptographic Processor Over GF(p)," *Circuits and Systems I: Regular Papers, IEEE Transactions*, vol. 53, no. 9, pp. 1946–1957, Sept. 2006.

[16] Gang Chen, Guoqiang Bai, and Hongyi Chen, "A High-Performance Elliptic Curve Cryptographic Processor for General Curves Over GF(p) Based on a Systolic Arithmetic Unit," *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS*, vol. 54, no. 5, pp. 412–416, May. 2007.

[17] Akashi Satoh and Kohji Takano, "A Scalable Dual-Field Elliptic Curve Cryptographic Processor," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 52, no. 4, pp. 449–460, April 2003.

[18] Yongyi Wu, Xiaoyang Zeng, "A new dual-field elliptic curve cryptography processor," *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium*, pp. 305–308, May 2006.

[19] Gutub A.A.-A, "VLSI core architecture for GF(p) elliptic curve crypto processor," *Electronics, Circuits and Systems, 2003. ICECS 2003. Proceedings of the 2003 10th IEEE International Conference*, vol. 1, pp. 84–87, Dec. 2003.

[20] Zhang Xiaopeng, Li Shuguo, "A High Performance ASIC Based Elliptic Curve Cryptographic Processor over GF(p)," *International Design and Test Workshop, 2007 2nd*, pp. 182–186, Dec. 2007.

[21] Bednara M., Daldrup M., Teich J., et al, “Tradeoff analysis of FPGA based elliptic curve cryptography,” *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium*, vol. 5, pp. V-797–V-800, May 2002.

[22] Wang You-Bo, Dong Xiang-Jun, Tian Zhi-Guang, “FPGA Based Design of Elliptic Curve Cryptography Coprocessor,” *Natural Computation, 2007. ICNC 2007. Third International Conference*, vol. 5, pp. 185–189, Aug 2007.

[23] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*. Cambridge, U.K.: Cambridge University Press, 1999, vol. 265, London Mathematical Society Lecture Note Series.

[24] K. Lauter, “The advantages of elliptic curve cryptography for wireless security,” *IEEE Wireless Commun.*, vol. 11, no. 1, pp. 62–67, Feb. 2004.

[25] Jia Xiangyu, Wang Chao, “The application of elliptic curve cryptosystem in wireless communication,” *MAPE 2005, IEEE International Symposium*, vol. 2, pp. 1602–1605, Aug. 2005.

[26] Mohammed E., Emarah A.E., El-Shennawy K., “Elliptic curve cryptosystems on smart cards,” Security Technology, 2001 IEEE 35th International Carnahan Conference, pp. 213–222, Oct. 2001.

[27] Malhotra K., Gardner S., Patz R., “Implementation of Elliptic-Curve Cryptography on Mobile Healthcare Devices,” Networking, Sensing and Control, 2007 IEEE International Conference, pp. 239–244, April 2007.

[28] Chi Huang, Jinmei Lai, Junyan Ren, et al, “Scalable elliptic curve encryption processor for portable application,” *ASIC 2003, Proceedings. 5th International Conference*, vol. 2, pp. 1312–1316, Oct. 2003.

[29] Ramachandran A., Zhibin Zhou, Dijiang Huang, “Computing Cryptographic Algorithms in Portable and Embedded Devices,” *Portable Information Devices, 2007. PORTABLE07. IEEE International Conference*, pp. 1–7, May 2007.

[30] Cilardo A., Coppolino L, Mazzocca N, Romano L, “Elliptic Curve Cryptography Engineering,” *Proceedings of the IEEE*, vol. 94, no. 2, pp. 395–406, Feb. 2006.

[31] IEEE 1363, *Standard Specifications for Publickey Cryptography*, 2000.

[32] NIST, Recommended elliptic curves for federal government use, May 1999.

http://csrc.nist.gov/encryption.

[33] Crowe F., Daly A., Marnane W., “A scalable dual mode arithmetic unit for public key cryptosystems,” *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference*, vol.1, pp. 568–573, April 2005.

[34] 徐茂智、游林编著，信息安全与密码学，清华大学出版社，2007 年 1 月.

[35] Diffie W., Hellman M., “New directions in cryptography,” *Information Theory, IEEE Transactions*, vol. 22, no. 6, pp. 644–654, Nov. 1976.

[36] NBS, “Data Encryption Standard,” FIPS Pub. 46, U.S. National Bureau of Standards, Washington DC, Jan. 1977.

[37] M. E. Hellman, “DES will be totally insecure within ten years,” *IEEE Spectrum*, vo1.16, no.7, pp. 32–39, July 1979.

[38] Seung-Jo Han, Heang-Soo Oh, Jongan Park, “The improved data encryption standard (DES) algorithm,” *Spread Spectrum Techniques and Applications Proceedings, 1996., IEEE 4th International Symposium*, vol. 3, pp. 1310–1314, Sept. 1996.

[39] R. Riverst, A. Shamir, L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[40] T. ElGamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” *IEEE Transactions on Information Theory*, vol. 31, pp. 469–472, 1985.

[41] 俞承杭主编，信息安全技术，科学出版社，2005 年.

[42] 管有庆、王晓军、董小燕编著，电子商务安全技术，北京邮电大学出版社，2005.

[43] S. Galbraith, S. Paulus, N. Smart, “Arithmetic on superelliptic curves,” *Mathematics of Computation*, vol. 71, pp. 393–405, 2002.

[44] 张先红编著，数字签名原理及技术，机械工业出版社，2004 年 1 月.

[45] J. Silverman, *The Arithmetic of Elliptic Curves*, Springer-Verlag, 1986.

[46] J. Silverman, *Advanced Topics in the Arithmetic of Elliptic Curves*, Springer-Verlag, 1994.

[47] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Applications*, Cambridge University Press, revised edition, 1994.

[48] L. Charlap and D. Robbins, *An elementary introduction to elliptic curves*, CRD Expository Report 31, Center for Communications Research, Princeton, 1988.

[49] L. Charlap and D. Robbins, *An elementary introduction to elliptic curves II*, CRD Expository Report 34, Center for Communications Research, Princeton, 1988.

[50] H. Cohen, *A Course in Computational Algebraic Number Theory*, Springer-Verlag, 1993.

[51] D. Knuth, *The Art of Computer Programming-Seminumerical Algorithms*, Addison-Wesley, 3rd edition, 1998.

[52] Schmidt-Samoa K., Semay O., Takagi T., "Analysis of fractional window recoding methods and their application to elliptic curve cryptosystems," *Computers, IEEE Transactions*, vol. 55, no. 1, pp. 48–57, Jan. 2006.

[53] Li Ming, Qin Baodong, Kong Fanyu, et al, "Wide-w-NAF Method for Scalar Multiplication on Koblitz Curves," *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007, SNPD 2007*, vol. 2, pp. 143–148, Aug. 2007.

[54] S. Pohlig and M. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Transactions on Information Theory*, vol. 24, pp. 106–110, 1978.

[55] J. Pollard, "Monte Carlo methods for index computation (mod p)," *Mathematics of Computation*, vol. 32, pp. 918–924, 1978.

[56] ANSI X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), American National Standards Institute, 1999.

[57] A. Atkin and F. Morain, "Elliptic curves and primality proving," *Mathematics of Computation*, vol. 61, pp. 29–68, 1993.

[58] G. Lay and H. Zimmer, "Constructing elliptic curves with givern group order over large finite fields," *Algorithmic Number Theory—ANTS-I, Lecture Notes in Computer Science, Springer-Verlag*, vol. 877, pp. 250–263, 1994.

[59] D. Johnson, Key validation, Research contribution to IEEE P1363, Available from http://grouper.ieee.org/groups/1363/Research, 1997.

[60] D. Johnson, Public Key validation: A piece of the PKI puzzle, Research contribution to IEEE P1363, Available from http://grouper.ieee.org/groups/1363/Research, 2000.

[61] D. Johnson, A. Menezes, S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, pp. 36–63, 2001.

[62] C. Koc, and C. Paar, "Cryptographic Hardwareand Embedded Systems–CHES'99," *Lecture Notes in Computer Science, Springer-Verlag*, vol. 1717, 1999.

[63] C. Koc and C. Paar, "Cryptographic Hardwareand Embedded Systems–CHES 2000," *Lecture Notes in Computer Science, Springer-Verlag*, vol. 1965, 2000.

[64] C. Koc, D. Naccache and C. Paar, "Cryptographic Hardwareand Embedded Systems–CHES 2001," *Lecture Notes in Computer Science, Springer-Verlag*, vol. 2162, 2001.

[65] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," *Advances in Cryptology—CRYPTO'99*, M. Wiener, Ed. Heidelberg, Germany: Springer-Verlag, Lecture Notes in Computer Science, vol. 1666, pp. 388–397, 1999.

[66] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," *Cryptographic Hardware and Embedded Systems—CHES 2001*, Ç. K. Koç, D. Naccache, and C. Paar, Eds. Heidelberg, Germany: Springer-Verlag, Lecture Notes in Computer Science, vol. 2162, pp. 251–261, 2001.

[67] J. J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and counter-measures for smard cards," *Smart Card Programming and Security (E-smart 2001)*, Heidelberg, Germany: Springer-Verlag, Lecture Notes in Computer Science, vol. 2140, pp. 200–210, 2001.

[68] I. Biehl, B. Meyer, and V. Muller, "Differential fault attacks on elliptic curve cryptosystems," *Advances in Cryptology—CRYPTO 2000*, M. Bellare, Ed.

Heidelberg, Germany: Springer-Verlag, Lecture Notes in Computer Science, vol. 1880, pp. 131–146, 2000.

[69] P. C. Kocher, “Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems,” *Advances in Cryptology—CRYPTO’96*, N. Koblitz, Ed. Heidelberg, Germany: Springer-Verlag, Lecture Notes in Computer Science, vol. 1109, pp. 104–113, 1996.

[70] A. Menezes, P. Van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[71] P. Barrett, “Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor,” *Advances in Cryptology—CRYPTO’86*, vol. 263, pp. 311–323, 1987.

[72] P. Montgomery, “Mondular multiplication without trial division,” *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[73] Stephen E. Eldridge and Colin D. Walter, “Hardware Implementation of Montgomery’s Modular Multiplication Algorithm,” *IEEE TRANSACTIONS ON COMPUTERS*, vo1.42, no.6, pp. 693–699, June 1993.

[74] C. Koc, T. Acar and B. Kaliski, “Analyzing and comparing Montgomery multiplication algorithm,” *IEEE Micro*, vol. 16, pp. 26–33, June 1996.

[75] Huapeng Wu, “Montgomery Multiplier and Squarer for a Class of Finite Fields,” *IEEE Transactions on Computers*, vo1.51, no.5, pp. 521–529, May 2002.

[76] McIvor C., McLoone M., McCanny J.V., “FPGA Montgomery modular multiplication architectures suitable for ECCs over GF(p),” *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium*, vo1.3, pp. 23–26, May 2004.

[77] McLoone M., McIvor C., McCanny J.V., “Montgomery modular multiplication architecture for public key cryptosystems,” *Signal Processing Systems, 2004. SIPS 2004*, pp. 349–354, 2004.

[78] FIPS 186-2, Digital Signature Standard (DSS), Federal Information Processing Standards Publication 186-2, National Institute of Standards and Technology, 1994.

[79] C. Lim and P. Lee, “More flexible exponentiation with precomputation,” Information Cryptology—*CRYPTO’94, Lecture Notes in Computer Science,* vo1.839, pp. 95–107, 1994.

[80] J. Lopez and R. Dahab, “High-speed software multiplication in GF(2^m),” *Progress in Cryptology—INDOCRYPT2000, Lecture Notes in Computer Science,* vo1.1977, pp. 203–212, 2000.

[81] Tao Zhou, Xingjun Wu, Guoqiang Bai, et al, “Fast GF(p) modular inversion algorithm
suitable for VLSl implementation,” *ELECTRONICS LETTERS*, vo1.38, no.14, pp. 706–707, July 2002.

[82] B. Kaliski, “The Montgomery inverse and its applications,” *IEEE Transactions on Computers*, vo1.44, pp. 1064–1065, 1995.

[83] Savas E., and Koc C.K., “The Montgomery modular inverse – revisited,” *IEEE Transactions on Computers*, vo1.49, no.7, pp.763–766, 2000.

[84] Adnan Abdul-Aziz Gutub, Alexandre Ferreira Tenca, Çetin Kaya Koç, “Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation,” *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI.02)*, pp. 46–51, April 2002.

[85] C. McIvor, M. McLoone and J.V. McCanny, “FPGA Montgomery modular multiplication architectures suitable for ECCs over GF(p),” *Circuits and Systems, 2004. ISCAS '04*, vo1.3, pp. 23–26, May 2004.

[86] C. McIvor, M. McLoone and J.V. McCanny, “Improved Montgomery modular inverse algorithm,” *ELECTRONICS LETTERS*, vo1.40, no.18, pp. 1110–1112, Sept. 2004.

[87] Guemc Meurice de Dormale, Philippe Bulens and Jean-Jacques Quisquater, “An Improved Montgomery Modular Inversion Targeted for Efficient Implementation on FPGA,” *ICFPT 2004*, pp. 441–444, 2004.

[88] Erkay Savas, “A Carry-Free Architecture for Montgomery Inversion,” *IEEE Transactions on Computers*, vo1.54, no.12, pp. 1508–1519, Dec. 2005.

[89] Yong-Jin Jeong and Wayne Burleson, “VLSI Array Synthesis for Polynomial

GCD Computation and Application to Finite Field Division," *IEEE Transactions on Circuits and Systems*, vol. 41, no. 12, pp. 891–897, Dec. 1994.

[90] J.H. Guo and C.L. Wang, "Hardware-Efficient Systolic Architecture for Inversion and Division in GF(2^m)," *Computers and Digital Techniques, IEE Proceedings*, vol. 145, no. 4, pp. 272–278, July 1998.

[91] Chien-Hsing Wu, Chien-Ming Wu, Ming-Der Shieh, et al, "High–speed low-complexity systolic designs of novel iterative division algorithms in GF(2^m)," *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 375–380, March 2004.

[92] A.K. Daneshbeh, M.A. Hasan, "A class of unidirectional bit serial systolic architectures for multiplicative inversion and division over GF(2^m)," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 370–380, March 2005.

[93] C. C. Wang, T. K. Truong, H. M. Shao, el al, "VLSI architectures for computing multiplications and inverses in GF(2^m)," *IEEE Trans.*, vol. C-34, no. 8, pp. 709–717, Aug. 1985.

[94] T. Itoh, S. Tsujii, "A fast algorithm for computing multiplicative inverses in GF(2^m) using normal basis," *Information of Computers*, vol. 78, no. 3, pp. 2140, Sept. 1988.

[95] G. L. Feng. "A VLSI architecture for fast inversion in GF(2^m)," *Computers, IEEE Transactions*, vol. 38, no. 10, pp. 1383–1386, Oct. 1989.

[96] N. Takagi, J. Yoshiki, and K. Takagi, "A Fast Algorithm for Multiplicative Inversion in GF(2^m) Using Normal Basis," *IEEE Trans. Computers*, vol. 50, no. 5, pp. 394–398, May 2001.

[97] Shuhua Wu, Yuefei Zhu, "A Resource Efficient Architecture for RSA and Elliptic Curve Cryptosystems," *Communications, Circuits and Systems Proceedings*, pp. 2356 – 2360, June 2006.

[98] S. B. Örs, L. Batina, B. Preneel, et al, "Hardware implementation of an elliptic curve processor over GF(p)," in *Proc. 14th IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP '03)*, pp. 433–443, June 2003.

[99] L Dupont, S Roy, J Y Chouinard, "A FPGA implementation of an elliptic curve cryptosystem," *ISCAS 2006*, May 2006.

[100] Sining Liu, F Bowen, B King, et al, “Elliptic curve cryptosystem implementation based on a look-up talbe sharing scheme,” *ISCAS 2006*, May 2006.

[101] William N. Chelton, Mohammed Benaissa, “Fast Elliptic Curve Cryptography on FPGA,” *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, vol. 16, no. 2, pp. 198–205, Feb. 2008.

[102] Hyun Min Choi, Chun Pyo Hong, Chang Hoon Kim, “High Performance Elliptic Curve Cryptographic Processor Over *GF*(2^{163}),” *4th IEEE International Symposium on Electronic Design, Test & Applications*, pp. 290–295, 2008.

致 谢

值此论文完成之际，我谨向我的导师盛世敏教授以及蒋安平副教授和于敦山高级工程师致以深深的谢意，衷心地感谢他们在这五年中对我的精心培养和关怀。盛老师渊博的知识、严谨的治学态度和高屋建瓴的学术思维，都使我在这五年的学习和研究工作中受益匪浅。蒋安平副教授和于敦山高级工程师认真的工作态度、活跃的思维方式和追求卓越的学术精神，也是我学习的榜样。正是由于三位老师在学术上对我的悉心指导和全力支持，我才能够顺利地开展本论文的工作，我工作中的每一个关键环节无不渗透着三位老师的指导和关注。

衷心感谢北京大学信息科学技术学院微电子学系 SoC 实验室和微电子所的老师们在我博士研究工作期间给我的大力支持和帮助。感谢吉利久教授、赵保英教授、倪学文教授、莫邦燹教授、张兴教授、陈中建副教授和贾嵩副教授在我的科研过程中给予我的很多指导意见。感谢韩临老师在 EDA 软硬件环境上给予我的大力支持和帮助。

由衷地感谢 SoC 实验室的所有同学们，在这几年的学习和生活中，大家互相关心、互相帮助，形成了很好的学术氛围。在此向 SoC 实验室的彭春干、张信、路卫军、李莹、叶乐、魏红雅以及已经毕业的李夏青、桂少华、陈佳、任芳、李刚、张师群、崔小欣、吴欣、范胜祥、赵悦汐、崔晓艳等同学表示感谢。还要感谢同在北京大学软硬件协同设计实验室工作和学习的陈江华、王展飞、李峰、谭志超等同学，他们在平常的生活、学习和科研中给了我很多有益的意见。

感谢我已经毕业的硕士班的所有同学以及博士班的所有同学，与大家在北大共同度过的岁月是每个人都应当珍惜的美好回忆。

感谢我的家人，是他们的关心和支持使我能够投入到学习和科研工作中去，是他们言传身教给我宝贵的人生哲理，使我能够在学业和生活中戒骄戒躁、充满自信地面对生活，完成学业。

最后，在这里我向所有关心、支持和帮助过我的人表示由衷的感谢！

博士期间论文发表情况

【期刊论文】

1、**王健**，蒋安平，盛世敏. 同时支持两种有限域的模逆算法及其硬件实现. 北京大学学报（自然科学版），第 43 卷，第 1 期，页码：138–143. 2007 年 1 月.

2、**王健**，盛世敏. 椭圆曲线加密体制的双有限域算法及其 FPGA 实现. 北京大学学报（自然科学版），第 44 卷，2008 年. **（已接收）**

【会议论文】

3、**Jian WANG** and AnPing JIANG, "A High-Speed Area-Efficient Architecture for the Arithmetic in GF(2^m)," 8th International Conference on Solid-State and Integrated Circuit Technology, pp. 2016–2018, Oct. 2006.

4、**Jian WANG** and AnPing JIANG, "An Area-Efficient Design for Modular Inversion in GF(2^m)," 2006 IEEE Asia Pacific Conference on Circuits and Systems, pp. 1498–1501, Dec. 2006.

5、**Jian WANG** and AnPing JIANG, "A High-Speed Dual Field Arithmetic Unit and Hardware Implementation," ASICON 2007, pp. 213–216, Oct. 2007.

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明： 所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名： 日期： 年 月 日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校□一年/□两年/□三年以后，在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名： 导师签名：

日期： 年 月 日