高速双域求逆单元的设计与实现*

蔡 亮[1]，戴紫彬[1]，陈 璐[2]
（1. 解放军信息工程大学 电子技术学院，河南 郑州 450004；
2. 武汉大学 计算机学院，湖北 武汉 430072）

摘 要：提出了一种能够在素数域和二进制域下，高速处理384位以下数据的模逆运算单元。其使用改进的Montgomery模逆算法将两个域下的求逆运算统一起来。设计中使用了S-C操作数分离表示法减少进位传播，并提出S-C减法运算修正算法保证其正确性。设计了双域S-C分离加减运算单元，使其能高速完成两个有限域下的算术运算。

关键词：双域；模逆；S-C分离表示；椭圆曲线；Montgomery求逆算法

The design and implementation of a high speed dual-field modular inversion unit

CAI Liang[1], DAI Zi Bin[1], CHEN Lu[2]
(1. Institute of Electronic Technology, The PLA Information Engineering University, Zhengzhou 450004, China;
2. School of Computer, Wuhan University, Wuhan 430072, China)

Abstract: This paper presents a modular inversion unit for prime and binary fields. Operands of the unit can be of any length no more then 384 bits. It uses modified montgomery inversion algorithm for these two finite fields. S-C operands separation algorithm has been used to decrease carry propagation. S-C subtraction correct algorithm has been proposed to ensure the correction of S-C operands separation algorithm.

Key words: dual-field; modular inverse; S-C separation; ECC; Montgomery inverse algorithm

目前，公钥密码加密系统以其便利的密钥分配管理、高度的安全性和易于实现数字签名算法等特性，逐渐在军用、商务政务等领域取得了广泛的应用。这使得国内外许多公司、大学甚至个人都对它产生了浓厚的研究兴趣。而椭圆曲线密码体系(ECC)又以其密钥长度小，密钥单位比特安全强度高、计算速度快等特点使其必将取代RSA算法，成为通用的公钥加密算法。

在椭圆曲线密码算法、Diffie-Hellma密钥分发协议和椭圆曲线数字签名算法(ECDSA)中，均包含求逆运算，而且二进制域和素数域下的求逆运算是椭圆曲线密码算法中最消耗时间的运算，因此，求逆运算的性能直接影响到整个系统的性能。

针对椭圆曲线密码算法所存在的问题，本文设计了一种支持二进制域和素数域的高速、低资源消耗的模逆运算，其占用1.7万逻辑门电路，在素数域下进行一次384位以下的模逆运算所需时间为9μs。

1 求逆算法

在椭圆曲线密码算法中，有二次将用到求逆运算：一次是在各点的仿射坐标经过坐标变换变为射影坐标进行点加或点倍运算后，还原到仿射坐标时。如选用标准系统，则有 $x_i=\frac{X_i}{Z_i}$、$y_i=\frac{Y_i}{Z_i}$，利用 X_i、Y_i 计算出结构后，需要乘以 Z_i^{-1}，来转换为 x_i、y_i 的值。这次求逆运算既可能是素数域，也可能是二进制域，其与椭圆曲线算法所选择的域有关；另一次求逆运算是在上层的数字签名算法中，签名运算时有些步骤需要计算取得随机整数 k 的模逆 $k^{-1}\bmod n$，而在验证签名时有些步骤需要计算接收到的 S 值的模逆 $W=S^{-1}\bmod n$。在椭圆曲线数字签名算法中的求逆运算都是素数域下的运算。

1.1 二进制域下的求逆算法

在二进制域下，加法与减法的运算都是“异或”操作，可以通过费马小定理[1]或扩展欧几里德定理来运算，

* 国家自然科学基金密码部件的设计自动化研究基金项目(60673071)；国家863项目基金可信PDA计算平台关键技术与原型系统研究基金项目(2006AA01Z442)

但是费马小定理在 $A^{-1}=A^{2(2^{m-1}-1)} \bmod F(x)$ 进行求逆运算时，模乘运算次数过多，导致系统性能下降。而改进的算法虽然可以减少模乘运算的次数，但其只能针对特定位数的操作数进行求逆，灵活性不强。

因此，在二进制域下，本设计选择了一种改进的 Montgomery 模逆算法[2]。该算法可以表示为：

输入：$a(x),p(x);\deg(a(x))<\deg(p(x))$

输出：$r(x),k$；$r(x)=a(x)^{-1}x^k \bmod p(x)$

$\deg(a(x))\le k\le \deg(p(x))+\deg(a(x))+1$

开始运算：

(1) $u(x)=p(x),v(x)=a(x),r(x)=0,s(x)=1$

(2) $k=0$

(3) while $v(x)\neq 0$,

(4) if $u(0)=0$, then $u(x)=u(x)/x,s(x)=x\cdot s(x)$

(5) else if $v(x)=0$, then $v(x)=v(x)/x,r(x)=x\cdot r(x)$

(6) else if $\deg(u(x))>\deg(v(x))$ then

$u(x)=(u(x)+v(x))/x$,

$r(x)=r(x)+s(x)$, $s(x)=x\cdot s(x)$

(7) else $v(x)=(v(x)+u(x))/x$

$r(x)=x\cdot r(x)$, $s(x)=s(x)+r(x)$

(8) $k=k+1$

(9) if $\deg(r(x))=\deg(p(x))$ then $r(x)=r(x)+p(x)$

(10) return $r(x)$ and k

此时给出的只是一个中间结果，需要根据用户需求对结果进行处理，以确定得到 Montgomery 域下的结果或一般域上的结果。如果需要得到 Montgomery 域下的结果，则需要将得到的 $r(x)$ 进行 Montgomery 乘法运算：$MonMult(r(x),x^{2n-k})=a(x)^{-1}\cdot x^k\cdot x^{2n-k}\cdot x^{-n}=a(x)^{-1}\cdot x^n$。式中，$n=\deg(p(x))$；如果要得到一般域上的 $a(x)^{-1}$，则还需要将 Montgomery 域下的结果再进行一步 Montgomery 乘法：$MonMult(a(x)^{-1},x^n,1)=a(x)^{-1}\cdot x^n\cdot x^{-n}=a(x)^{-1}$ 后得到最终结果[3]。

1.2 素数域下的求逆算法

素数域下的求逆算法也包括利用费马小定理和扩展欧几里德两种，但是在素数域下，大数的模乘和模平方消耗的时间和资源更大，因此利用费马小定理求素数域上的模逆更加不现实。

在素数域下，依然可以采用改进的 Montgomery 算法进行模逆运算：

输入：$a\in[1,p-1]$ 和 p

输出：$r\in[1,p-1],k,r=a^{-1}2^k \bmod p$ 且 $n\le k\le 2n$

开始运算：

(1) $u=p$, $v=a,r=0,s=1$

(2) $k=0$

(3) while $v\neq 0$,

(4) if u 为偶数, then $u=u/2$, $s=2\cdot s$

(5) else if v 为偶数, then $v=v/2$, $r=2\cdot r$

(6) else if $u>v$ then $u=(u-v)/2$, $r=r+s$, $s=2\cdot s$

(7) else $v=(v-u)/2$, $r=2\cdot r$, $s=s+r$

(8) $k=k+1$

(9) if $r\ge p$ then $r=r-p$

(10) return $r=p-r$ 和 k

与二进制域相同，此时得到的也仅仅是一个中间结果，需要进行相应的 Montgomery 乘法来得到所需的结果。如果需要得到 Montgomery 域下的结果，则需要将得到的 r 进行素数域上 Montgomery 乘法运算：$MonMult(r,2^{2n-k})=a^{-1}\cdot 2^k\cdot 2^{2n-k}\cdot 2^{-n}=a^{-1}\cdot 2^n$；如果要得到一般域上的 a^{-1}，则还需要将 Montgomery 域下的结果再进行一步 Montgomery 乘法：$MonMult(a^{-1},2^n,1)=a^{-1}\cdot 2^n\cdot 2^{-n}=a^{-1}$。

1.3 双域下的求逆算法

从两个域下的求逆算法可以看出，两个域下的求逆算法的结构基本相同，只是中间的基本步骤有些不同，因此只需要将某些相应的处理单元设计成支持双域的，就可以依据统一结构设计求逆单元。

二进制域与素数域上相统一，能够共用实现单元的运算如下：

(1) 二进制域下的 $u(x)/x$ 运算与素数域下的 $u/2$ 运算，都可以通过循环右移一位完成。

(2) 二进制域下的 $x\cdot s(x)$ 与素数域下的 $2\cdot r$ 运算方式也是统一的，均能通过循环左移一位完成。

(3) 素数域下的判断奇偶运算与二进制域下判断最低位是否为 0 对应。

两种求逆算法中，相同步骤其实现方式不同，需要分别设计的部分有：

(1) 二进制域与素数域下的加减法不同，二进制域下的加减运算无论加减均为按位“异或”。而素数域下的加减法却包含进位传播。

(2) 在两个算法的第 6 步，判断 u、v 及 $\deg(u(x))$、$\deg(v(x))$ 大小时，素数域上必须通过判断 u、v 做减法后的进位标志位，而二进制域下判断 $\deg(u(x))$、$\deg(v(x))$ 的大小则需要将 $u(x)$、$v(x)$ 按位“异或”。判断结果中最高位为 1 的位置，假设为第 n 位，如果 $u(x)$ 的第 n 位为 1，则有 $\deg(u(x))>\deg(v(x))$，如果 $v(x)$ 的第 n 位为 1，则有 $\deg(v(x))>\deg(u(x))$；如果“异或”结果全为 0，则 $u(x)=v(x)$。这是因为，在二进制域下，两个数的大小实际上是从高位到低位进行比较，一次“异或”操作可以将两个操作数最高的相异位确定出来，该位为 1 的数即为较大的数。

综上所述，素数域和二进制域下的求逆运算中，相同的运算结构可以复用。对如加减法这种不同的运算，则需要设计双域的加减法器，添加域选择信号时选择对应的模块，使求逆运算的流程更加统一，便于实现。

2 硬件结构

2.1 双域算术运算结构

素数域与二进制域下加减法运算的主要区别在于进位传播。二进制域下 $a\pm b$ 均为 a xor b，素数域下为了避免进位传播，与二进制域的运算须统一结构，可以选

择三进二出的 CSA 加法器。如要计算 $x+y$ 的值，输入为 x、y、z，z 可以看作为加减运算选择信号 add/sub，$z=0$，输出结果 $s=x$ xor y xor z，$c_{i+1}=(x_i$ and $y_i)$or $(y_i$ and $z_i)$or$(z_i$ and $x_i)$。x、y 的和等于 s 与 c 进行全加的和。如要计算 $x-y$ 的值，只需要将输入改为 x、$\bar{y}$、z，$z=1$。将得到的 s 与 c 进行全加以后去掉最高标志位，即为减法结果。

在素数域的求逆算法中，在第 6、7、9、10 步中用到了减法操作，而在 6、7 步中，减法操作后接着是一个除 2 操作；在第 6、7、8 步中用到了加法操作。如果每一步都将加减法的结果计算出来，则随着操作数位数的增加，进位传播的耗时会越来越长，使得系统性能大大降低。因此，可以考虑在进行这些加减操作时，采用 CSA 加法器进行运算，得到的结果不进行全加，直接带入下一步进行运算，只是在计算最终结果时进行一步全加操作。这样，就可以节约大量的时间，但是，也必须对算法进行修改，以解决出现的新问题。

CSA 加法器的加法运算和普通的进位传播加法器是统一的，s 与 c 的和即为最后结果。但是在进行减法运算时，$x-y=x+\bar{y}+1$，用全加器实现 n 位减法实际上是将第 $n+1$ 位的进位舍去了，如果 $x>y$，得到的结果一定小于 x。如果使用 CSA 加法器来进行 $x-y$ 运算，则三个输入为 x、$\bar{y}$、1，其中，如果 x、$\bar{y}$ 为 n 位，输出的 s 也为 n 位，c 位为 $n+1$ 位。如果将 s 与 c 用 CPA 加法器进行全加，舍去第 $n+1$ 位的结果，则结果正常。如采取 $s-c$ 形式，不将和进行全加，将 3 个 n 位的数相加只会出现如表 1 所示的四种情况。

表 1 $s-c$ 减法修正表

类型	S_n 的值	C_n 的值	$(S+C)_n$ 的值	对 s、c 的修正
1	0	0	0	不操作
2	0	0	1	-2^n
3	0	1	1	-2^{n+1}
4	0	1	0	-2^n

表 1 中，$(s+c)_n$ 相当于标准减法器的进位标志位，是在减法运算中需要舍去的部分。但由于 s 和 c 并没有进行全加，$(s+c)_n$ 还隐性地包含于 s 和 c 中，使得使用全加器求该标志位的进位传播很长，但超前进位加法器中的进位逻辑可以在一个时钟内得到：

$$
\begin{aligned}
&(s_{n-1} \text{ and } c_{n-1})\\
&\text{or}(s_{n-2} \text{ and } c_{n-2}) \text{ and } (s_{n-1} \text{ or } c_{n-1})\\
&\text{or}(s_{n-3} \text{ and } c_{n-3}) \text{ and } (s_{n-2} \text{ or } c_{n-2}) \text{ and } (s_{n-1} \text{ or } c_{n-1})\\
&(s+c)_n=\\
&\qquad\vdots\\
&\text{or}(s_1 \text{ and } c_1) \text{ and } (s_2 \text{ or } c_2) \text{ and } (s_{n-2} \text{ or } c_{n-2}) \text{ and } (s_{n-1} \text{ or } c_{n-1})\\
&\text{or}(s_0 \text{ and } c_0) \text{ and } (s_1 \text{ or } c_1) \text{ and } (s_{n-2} \text{ or } c_{n-2}) \text{and } (s_{n-1} \text{ or } c_{n-1})
\end{aligned}
$$

从表 1 中可以看到：s_n 始终为 0，当 c_n 为 0、$(s+c)_n$ 为 0 时，说明减法运算的进位标志位为 0，s 与 c 的和无需修正就能代表减法运算的结果；如果 c_n 为 0、$(s+c)_n$ 为 1 或者 c_n 为 1、$(s+c)_n$ 为 0 时，说明减法运算的进位标志位为 1，需要在结果中减去 2^n 这个溢出值。但这两种情况也有一些区别：如果 c_n 为 1、$(s+c)_n$ 为 0 时，直接去掉 c_n，则相当于减去了 2^n，剩下的 n 位 s、c 的和即为位减法运算的结果，s 与 c 也可以直接带入后面的运算步骤中进行运算；如果 c_n 为 0、$(s+c)_n$ 为 1，需要减去 2^n，但是 c_n 为 0，即 $s<2^n$，$c<2^n$，但 $s+c>2^n$，此时为了修正结果只有将 2^n 分摊到 s 与 c 中消去。本文设计了一个在 s 与 c 中，不进行全加而消去 2^n 的算法，如下：

输入：两个 n bit 数 s 与 c，其中 $s<2^n$，$c<2^n$，且 $s+c>2^n$；

输出：两个 n bit 数 s' 与 c'，$s'+c'=s+c-2^n$。

(1) s 与 c 做按位“与”操作 $m=s$ and c；

(2) $i=n-1$；

(3) While$(m_i\neq 1)$ $i=i-1$ End while；

设 $m_k=1$ 跳出循环

(4) $m_k:=0$，$m[k-1$ downto $0]:=1$；

(5) $s'=s$ and m，$c'=c$ and m。

该算法的原理是将需要减去的 2^n 变形为：

$2^n=2^{n-1}+2^{n-1}=2^{n-1}+2^{n-2}+2^{n-2}=2^{n-1}+2^{n-2}+2^{n-3}+2^{n-3}=\cdots$，

s 和 c 要产生进位，就必须有进位产生位 $s_k=c_k=1$；而且要保证进位能够顺利地传递到最高位，即从最高位 $n-1$ 位到第 $k+2$ 位，s or c 均为 1。也就是说，从最高位到第 $k+1$ 位，$s_k=c_k=1$ 之间总有 $s_i=1$ 或者 $c_i=1$。如果将第 $k+1$ 位的两个 1 及其高位的所有 1 均置 0 的话，$s+c$ 实质上减少了 $2^{n-1}+2^{n-2}+2^{n-3}+\cdots+2^k+2^{k-1}+2^{k-1}=2^n$ 达到了隐性减去 2^n 的目的。反映在算法中就是先将 m 的 $k+1$ 位到 $n-1$ 位均置为 0，k 位到第 1 位均置为 1；再将 m 分别与 s 和 c 相“与”得到输出结果 s'、c'。如果 c_n 为 1，$(s+c)_n$ 也为 1，则需要先将 c_n 去掉，再利用算法 1 对 s 和 c 进行修正。但这种情况出现比较少。

修正算法实现起来结构简单，唯一耗时的运算就是寻找 m 中最高位 1 的位置，其他的运算均能通过组合逻辑完成。逐位扫描与全加运算相比较，其性能可以提高 60%。而将 s 和 c 分离表示避免全加运算可以在每一轮减法运算的时候都节省出进位传播的时间，这样累计起来，性能的提升相当可观。

2.2 $s-c$ 分离表示求逆的硬件结构

采用 $s-c$ 分离 CSA 进行加减法运算需要对 $s-c$ 值进行处理，因此结构要比常规加减法复杂。双域加减运算的结构如图 1 所示。

首次输入时，$x_{(c)}$、$y_{(c)}$ 均为 0。field 信号控制域的选择，如果为二进制域，则 c 位的输出都为 0，在 CSA2 后就可以输出加减的结果；如果 field 信号控制域选择了素数域，在进行加法时，s 和 c 的输出不必经过修正电路，并且在 CSA2 后能得到加法运算的结果；如果进行素数

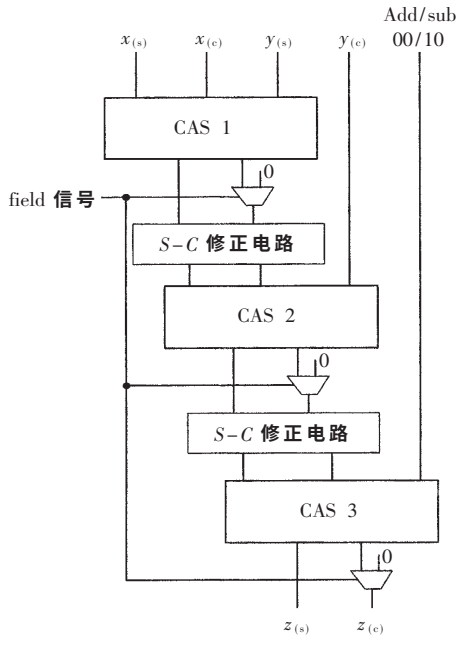

图 1 S-C 分离双域加减逻辑

域下的减法运算，则相当于计算 $x_{(s)}+x_{(c)}+\bar{y}_{(s)}+1+\bar{y}_{(c)}+1$，在经过CSA2 加法器后，还必须加上二进制数 10，并且 CSA1 和 CSA2 的输出数据都要进行 s-c 溢出修正，以保证最终输出的结果 $z_{(s)}$、$z_{(c)}$能够代表计算结果，不会存在进位问题。

整个双域求逆运算单元的结构如图 2 所示。U、V、R、S 寄存器输出的值皆以 s-c 形式表示。算法第 4、5 步可以并行执行，因此，U 和 V 的值进入 double/half[A]、R 和 S 的值进入 double/half[B]，在第 4 步时同时对 U 和 S 进行处理，在第 5 步时同时对 V 和 R 进行处理。

3 综合与结果分析

模逆操作的时间取决于操作数上的 0、1 分布，因此时间是不确定的。平均情况下模逆运算所需周期的计算公式为：对一周期操作 w 位的设计，模逆运算的平均周期数为 $C=(2.4124n+1)[\frac{n}{w}]$[4]。

整个设计使用 Verlog 语言描述，采用 Synopsys 公司的 Design Complier 在 SIMC 0.18μm-typcal 工艺库下综合，等效与非门为 1.7 万门，最高工作频率可达300MHz，完成一次 384bit 的 $GF(p)$求逆只需要 9μs。表 2 是本文模乘器与已发表文献中同类设计的比较结果。

表 2 求逆运算性能比较表

设计	适应域	操作数长/bit	运算时间/μs	时间频率/MHz
参考文献[2]	$GF(p)$	160 192	148 205	450
参考文献[4]	双域	160	180	5
参考文献[4]	双域	160	60	50
参考文献[5]	双域	0~512	16(160bit)	300
本文	双域	0~384	9(394bit)	300

本文提出了一种高速双域求逆解决方案，统一了二进制域和素数域上求逆算法流程。中间计算结果采用 s-c 分离表示法节省了大量的进位传播时间，设计了通用的双域加减单元节约了面积，使得该求逆运算模块在性能上比同类设计有很大提高。

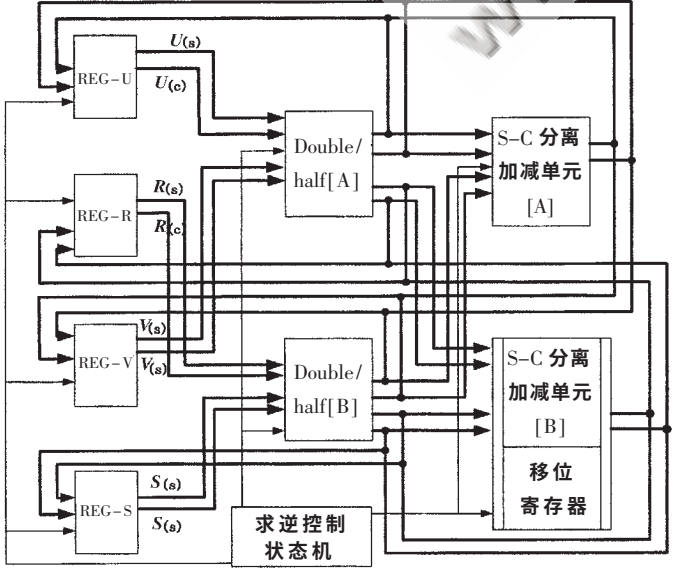

图 2 双域运算单元结构

参考文献

[1] KALISKI B S. The montgomery inverse and its applications. IEEE. Transactions on Computers, 1995,44(8):1064-1065.

[2] SAVAS E, KOC C K. The montgomery modular inverse-revisited. IEEE, Transactions on Computers, 2000,49(7):763-766.

[3] MCIVOR C J, MCLOONE M, MCCANNY J V. Improved montgomery modular inverse algorithm[J]. IEEE Electronics Letters, 2004,40(18).

[4] KOC G T. Scalable VLSI architecture for GF(p) Montgomery modular inverse computation. ISVLSI 2002 IEEE Computer Society Annum Symposium on VLSI, Pittsburgh, Pennsylvania, 2002:25-26.

(收稿日期：2008-01-25)