A fast multiplier over GF (2^n)

A Fast Multiplier over $GF(2^n)$

M.J. Potgieter[1], B.J. van Dyk[1] and Tj.J. Tjalkens[2]

[1] University of Pretoria, Pretoria, South Africa
[2] Eindhoven University of Technology, Eindhoven, the Netherlands
e-mail: t.j.tjalkens@tue.nl

Abstract. In this paper we will present a hardware implementation of a $GF(2^n)$ polynomial basis multiplier that is twice as fast a the classical multiplier while requiring about 50 % more chip area. We implement a flexible scalar (or point) multiplier for elliptic curve cryptosystems using this multiplier and find that the flexible system performs almost twice as fast as compared with the classical multiplier.

1 Introduction

An elliptic curve cryptosystem (ECC) can be used to exchange keys over an insecure channel. ECC belongs to the class of *public key cryptosystems.* The famous Diffie-Hellman system [1] relies on the prohibitive complexity of solving s from

$$g^s \equiv k \bmod p, \tag{1}$$

where g, k, an p (a large prime) are known. The RSA system [2] is a widely used variant on this theme.

In an ECC another complex operation is used, namely the *point addition*, see [3, 4] or the more recent [5]. Here solving m from mP, where P is a known point on a known elliptic curve in $GF(2^n)$ is the intractable operation. The operation mP is known as the *elliptic curve scalar multiplication.*

For the Diffie-Hellman and RSA systems that use a multiplicative group, a sub-exponential, w.r.t. $\log p$, running time algorithm, the *index-calculus method*, exists, while all known algorithms for solving the EC scalar multiplication problem are still exponential in n. Thus, much smaller keys can be used in an ECC (about 160 bit keys) than in RSA systems (about 1000 bit keys). Even though the basic operation in an ECC, point addition, might be more complex than the multiplication in RSA, the difference in field sizes needed, make the ECC system an attractive choice for low power/low complexity applications.

While implementing a flexible, programmable, hardware EC scalar multiplier we found a novel field multiplier that is twice as fast as the standard multiplier, while requiring 50% more chip area.

Because we are interested in a flexible system a polynomial basis for $GF(2^n)$ seems more appropriate than a normal basis. Multiplication in a normal basis representation is only efficient if an optimal normal basis exists, and in the range of n that we were interested in, 160...200, only a few optimal bases exists.

So, the field elements are expressed as binary vectors $(a_0, a_1, \ldots, a_{n-1})$ of dimension n, relative to the base $\{1, \alpha, \alpha^2, \ldots, \alpha^{n-1}\}$, where α is a root of the irreducible polynomial $f(x)$ of degree n over $\mathbb{F}_2$.

The *classical multiplier* implements the multiplication operation together with the modular reduction. The following pseudo-code describes this multiplier and figure 1 gives a hardware implementation hereof. It is clear that the running time, or number of clock cycles, is equal to n. Note that the multiplication by x in the code is a simple *left shift* of the register r.

Inputs: a, b and the polynomial $f(x)$.
Output: r, where $r \equiv a \cdot b \bmod f(x)$.

$r \leftarrow 0$
for i **from** n **downto** 0 **do**
  $msb \leftarrow r_{n-1}$
  $r \leftarrow (msb$ **and** $f(x)) \oplus (r * x)$
  $r \leftarrow r \oplus (a_i$ **and** $b)$
endfor
return r

Listing 1: Pseudo code for the classical multiplier

For odd characteristic fields, a more efficient multiplier exists, the Montgomery multiplier [6]. Applying this idea to $GF(2^n)$ we end up with a multiplier that is very similar to the classical multiplier, see Listing 2. The main difference is that it operates on the least significant bits of a.

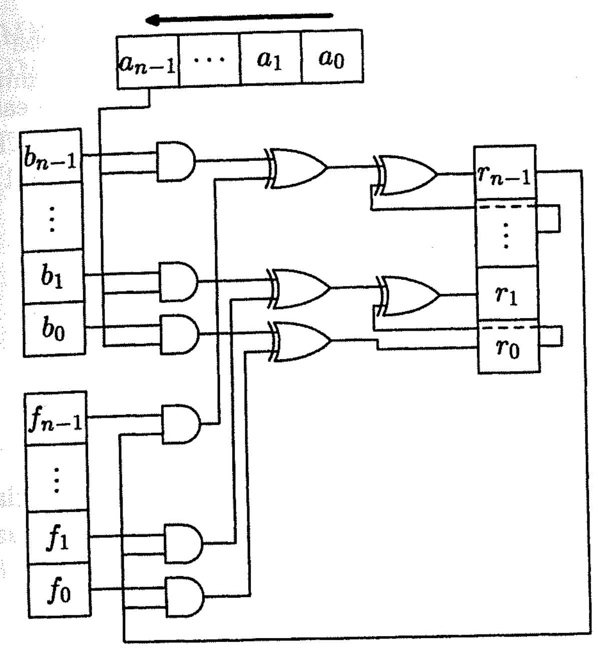

Fig. 1. The classical multiplier

The running time is again equal to n clock cycles. Note also that the division by x in the code is a simple *right shift* of the register r.

Inputs: a, b and the polynomial $f(x)$.
Output: r, where $r \equiv a \cdot b \cdot x^{-n} \bmod f(x)$.

$r \leftarrow 0$
for i **from** 0 **to** $n - 1$ **do**
  $r \leftarrow r \oplus (a_i$ **and** $b)$
  $r \leftarrow (r_0$ **and** $f(x)) \oplus r$
  $r \leftarrow r/x$
endfor
return r

Listing 2: Pseudo code for the Montgomery multiplier

Instead of computing $r \equiv a \cdot b \bmod f(x)$ the Montgomery multiplier (MM) computes $MM[a,b] \triangleq r \equiv a \cdot b \cdot x^{-n} \bmod f(x)$. Therefor we represent every element $a \in GF(2^n)$ by $M(a) \triangleq a \cdot x^n \bmod f(x)$. So, if we wish to compute $a \cdot b \bmod f(x)$ we compute $MM[M(a), M(b)] \equiv a \cdot b \cdot x^n \bmod f(x)$. We observe that $MM[M(a), M(b)] = M(ab)$. Converting a value a to its Montgomery representation $M(a)$ is easily obtained using the Montgomery multiplier as $M(a) = MM[a, x^{2n}]$. The conversion from $M(a)$ to a is similarly performed by $a = MM[M(a), 1]$.

3 A modified field multiplier

If we briefly ignore the modular reduction we see that the classical multiplier and the Montgomery multiplier actually compute the same product. Say $a(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ and $b(x) = b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$. Then the classical multiplier results in

$$(a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0)(b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0) = c_8x^8 + c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \quad (2)$$

It is easy to see that the result for the Montgomery multiplier is $c(x) \cdot x^{-n}$ because the intermediate results are shifted to the right.

a_4b_4	a_4b_3	a_4b_2	a_4b_1	a_4b_0				
	a_3b_4	a_3b_3	a_3b_2	a_3b_1	a_3b_0			
		a_2b_4	a_2b_3	a_2b_2	a_2b_1	a_2b_0		
			a_1b_4	a_1b_3	a_1b_2	a_1b_1	a_1b_0	
				a_0b_4	a_0b_3	a_0b_2	a_0b_1	a_0b_0
c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

Assume that n is even, then multiplying $c(x)$ by $x^{n/2}$ can be seen as a combination of left and right shifts on the partial results. However these results can be performed in parallel and the two partial end results

can then be added together. It is still possible to perform the modular reduction using the original algorithms from listing 1 and listing 2.

Inputs: a, b and the polynomial $f(x)$.
Output: r, where $r \equiv a \cdot b \cdot x^{-n/2} \bmod f(x)$.

$rc \leftarrow 0$
$rm \leftarrow 0$
for i **from** 0 **to** $n/2 - 1$ **do**
  $msb \leftarrow rc_{n-1}$
  $rc \leftarrow (msb$ **and** $f(x)) \oplus (rc * x)$
  $rc \leftarrow rc \oplus (a_{n-1-i}$ **and** $b)$
  $rm \leftarrow rm \oplus (a_i$ **and** $b)$
  $rm \leftarrow (rm_0$ **and** $f(x)) \oplus rm$
  $rm \leftarrow rm/x$
endfor
return $rc \oplus rm$

Listing 3: Pseudo code for the modified multiplier

The main increase in chip area is caused by the extra (temporary) result register, rm and rc in stead of r in the original multipliers.

It is easy to extend this method to the case where n is odd. In that case the Montgomery part works on the $(n-1)/2$ least significant coordinates of a and the representation is $M(a) = a \cdot x^{(n-1)/2} \bmod f(x)$. The number of clock cycles needed is $(n+1)/2$. Odd, or preferably prime, values for n are often used in cryptographic systems.

4 Comparing the classical and modified multipliers

The multipliers were implemented in a *field programmable gate array* FPGA. The following two tables give an indication of the speed-up and chip area cost of the modified multiplier relative to the classical multiplier. We show the timing and chip area for the complete ECC scalar multiplier unit in table 1 and table 2 respectively.

5 Conclusion

The results of the ECC scalar multiplier comparison indicate that the field multiplier is the predominant factor in the speed-up of the design.

Field size n	$t_{\text{classical}}$	t_{modified}
163	6.619	3.776
233	13.316	7.158
283	19.518	10.299

Table 1. Timing comparison in milliseconds

Field size n	% slices (modified)	% slices (classical)	% modified / % classical
96	35	26	1.346
192	56	40	1.400
304	81	56	1.446
384	99	67	1.478

Table 2. Chip area in FPGA slices

The increase in chip area is caused by the field multiplier only. This and a more detailed comparison of the designs show that the modified multiplier costs about 50 % more chip area. Because we designed a flexible and programmable ECC unit, many optimizations that are possible with fixed and clever choice of parameters were not possible in this implementation. This might influence the speed and chip area cost of a design enormously. Still the speed-up factor will be more or less the same because in many designs the field multiplier will determine the speed of the overall circuit.

References

1. Diffie, W. and M.E. Hellman, "New directions in cryptography," *IEEE Trans. Inform. Theory*, vol. 22, 1976, pp. 644–654.
2. Rivest, R.L., A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," Comm. ACM, vol. 21, 1978, pp. 120–126.
3. Koblitz, N., "Elliptic curve cryptosystems," *Math. Comp.*, vol 48, 1987, pp. 203–209.
4. Miller, V., "Use of elliptic curves in cryptography", In *Advances in Cryptology, CRYPTO 85*, Ed. H.C. Williams, Springer-Verlag, LNCS 218, 1986, pp. 417–426.
5. Blake, I.F., G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, Cambridge, 1999, pp. 1–76.
6. Montgomery, P.L., "Modular multiplication without trial division," *Math. Comp.*, vol 44, 1985, pp. 519–521.