Chapter 18

Division in *GF(p)* for Application in Elliptic Curve Cryptosystems on Field Programmable Logic

Alan Daly[1], William Marnane[1], Tim Kerins[1], and Emanuel Popovici[2]

[1] *Department of Electrical & Electronic Engineering, University College Cork, Ireland*
Email: {aland,liam,timk}@rennes.ucc.ie

[2] *Department of Microelectronic Engineering, University College Cork, Ireland*
Email: e.popovici@ucc.ie

Abstract Elliptic Curve Cryptosystems (ECC) are becoming increasingly popular for use in mobile devices and applications where bandwidth and chip area are limited. They provide much higher levels of security per key bit than established public key systems such as RSA. The core ECC operation of point scalar multiplication in $GF(p)$ requires modular multiplication, division/inversion and addition/subtraction. Division is the most costly operation in terms of speed and is often avoided by performing many extra multiplications. This paper proposes a new divider architecture and FPGA implementations for use in an ECC processor.

18.1 Introduction

Elliptic Curve Cryptosystems (ECC) were independently proposed in the mid-eighties by Victor Miller [Miller, 1985] and Neil Koblitz [Koblitz, 1987] as an alternative to existing public key systems such as RSA and DSA. No sub-exponential algorithm is known to solve the discrete logarithm problem on a suitably chosen elliptic curve, meaning that smaller parameters can be used in ECC with equivalent security to other public key systems. It is estimated that an elliptic curve group with 160-bit key length has security equivalent to RSA with a key length of 1024-bit [Blake et al., 2000].

Two types of finite field are popular for use in elliptic curve public key cryptography: *GF(p)* with p prime, and $GF(2^n)$ with n a positive integer. Many implementations focus on using the field $GF(2^n)$ due to the underlying arithmetic

P. Lysaght and W. Rosenstiel (eds.),
New Algorithms, Architectures and Applications for Reconfigurable Computing, 219–229.
© 2005 *Springer. Printed in the Netherlands.*

which is well suited to binary numbers [Ernst et al., 2002]. However, most *GF*(2^n) processors are limited to operation on specified curves and key sizes. An FPGA implementation of a *GF*(2^n) processor which can operate on different curves and key sizes without reconfiguration has previously been presented in [Kerins et al., 2002]. ECC standards define different elliptic curves and key sizes which ECC implementations must be capable of utilising [IEEE, 2000]. With a *GF(p)* processor, any curve or key length up to the maximum size p, can be used without reconfiguration.

Few implementations of ECC processors over *GF(p)* have been implemented in hardware to date due to the more complicated arithmetic required [Orlando and Paar, 2001].

The modular division operation has been implemented in the past by modular inversion followed by modular multiplication. No implementations to date have implemented a dedicated modular division component in an ECC application.

This paper proposes a modular divider for use in an ECC processor targeted to FPGA which avoids carry chain overflow routing. The results presented include divider bitlengths of up to 256-bits, which would provide security well in excess of RSA-1024 when used in an ECC processor.

18.2 Elliptic Curve Cryptography over $GF(p)$

An elliptic curve over the finite field *GF(p)* is defined as the set of points (x, y), which satisfy the elliptic curve equation

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

where x, y, a and b are elements of the field and $4a^3 + 27b^2 \neq 0$.

To encrypt data, it is represented as a point on the chosen curve over the finite field. The fundamental encryption operation is point scalar multiplication, i.e. a point P (x_P, y_P) is added to itself k times, to get point Q (x_Q, y_Q).

$$\begin{aligned} Q &= kP \\ &= \underbrace{P + P + \cdots + P}_{k\ times} \end{aligned}$$

Recovery of k, through knowledge of the Elliptic Curve equation, base point P, and end point Q, is called the Elliptic Curve Discrete Logarithm Problem (ECDLP) and takes fully exponential time to achieve. The Elliptic Curve Diffie-Hellman Key Exchange protocol uses this fact to generate shared secret keys used for bulk data encryption.

In order to compute kP, a double and add method is used and k is represented in binary form and scanned right to left from LSB to MSB, performing a double

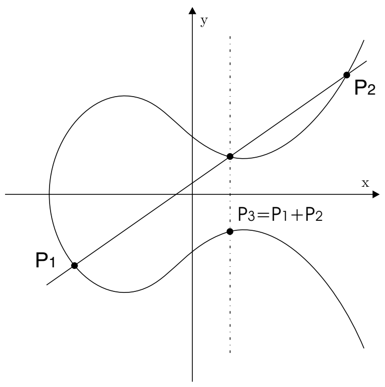

Figure 18.1. Point Addition.

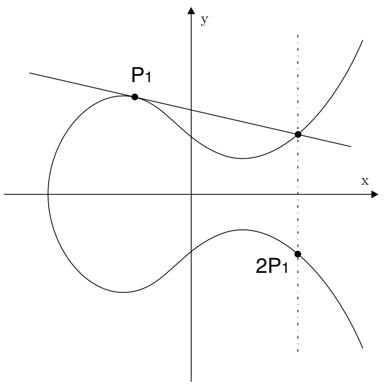

Figure 18.2. Point Doubling.

at each step and an addition if k_i is 1. Therefore the multiplier will require $(m - 1)$ point doublings and an average of $(\frac{m-1}{2})$ point additions, where m is the bitlength of the field prime p.

The operations of elliptic curve point addition and point doubling are best explained graphically as shown in Fig. 18.1 and Fig. 18.2. To add two distinct points, P_1 and P_2, a chord is drawn between them. This chord will intersect the curve at exactly one other point, and the reflection of that point through the x-axis is defined to be the point $P_3 = P_1 + P_2$.

In the case of adding point P_1 to itself (doubling P_1), the tangent to the curve at P_1 is drawn and found to intersect the curve again at exactly one other point. The reflection of this point through the x-axis is defined to be the point $2P_1$. (Note: The *point at infinity*, $\mathcal{O}$ is taken to exist infinitely far on the y-axis, and is the identity of the elliptic curve group.)

The point addition/doubling formulae in *affine coordinates* are given below. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then $P_3 = (x_3, y_3) = P_1 + P_2$ is given by:

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$
$$\lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2 \\ \dfrac{3x_1^2 + a}{2y_1} & \text{if } P_1 = P_2 \end{cases}$$

Different coordinate systems can be used to represent the points on an elliptic curve, and it is possible to reduce the number of inversions by representing the points in *projective coordinates*. However, this results in a large increase in the number of multiplications required per point operation (16 for addition, 10 for doubling) [Blake et al., 2000].

Processors using affine coordinates require far fewer clock cycles and hence less power than processors based on projective coordinates [Orlando and Paar, 2001]. Point addition requires 2 multiplications, 1 division and 6 addition/subtraction operations. Point doubling requires 3 multiplications, 1 division and 7 additions/subtractions. The overall speed of an affine coordinate-based ECC processor will hence rely on an efficient modular division operation.

18.3 Modular Inversion

A common method to perform the division operation is to perform an inversion followed by a multiplication.

The multiplicative inverse of an integer $a \pmod M$ exists if and only if a and M are relatively prime. There are two common methods to calculate this inverse. One is based on Fermat's theorem which states that $a^{M-1} (\text{mod } M) = 1$ and therefore $a^{M-2} (\text{mod } M) = a^{-1} \pmod M$. Using this fact, the inverse may be calculated by modular exponentiation. However this is an expensive operation requiring on average $1.5 \log_2 m$ multiplications.

Kaliski proposed a variation of the extended Euclidean algorithm to compute the Montgomery inverse, $a^{-1}2^m \pmod M$ of an integer a [Kaliski, 1995]. The output of this algorithm is in the Montgomery domain, which is useful when performing further Montgomery multiplications. The Montgomery multiplication algorithm is an efficient method to perform modular multiplication without the need for trial division. When many multiplications are necessary, its benefits are fully exploited by mapping inputs into the "Montgomery Domain" before multiplication, and re-mapping to the integer domain before output. This is especially useful in applications such as the RSA cryptosystem where modular exponentiation is the underlying operation. Efficient Montgomery multipliers for implementation on FPGA have previously been presented in [Daly and Marnane, 2002].

Kaliski's method requires a maximum of $(3m + 2)$ clock cycles to perform an m-bit Montgomery inversion, or $(4m + 2)$ cycles to perform a regular integer inversion. Montgomery inverters have previously been presented in [Gutub et al., 2002][Daly et al., 2003][Savas and Koc, 2000].

18.4 Modular Division

A new binary add-and-shift algorithm to perform modular division was presented recently by S.C. Shantz in [Shantz, 2001]. This provides an alternative to division by inversion followed by multiplication. The algorithm is given here in algorithm 1. It computes $U = \frac{y}{x} \pmod M$ in a maximum of $2(m - 1)$ clock cycles where m is the bitlength of the modulus, M.

Algorithm 1 : Modular Division

Input : $y,\ x\ \in [1, M-1]$ and M
Output : U, where $y\ =\ U * x \pmod{M}$

```
01. $A := x, B := M, U := y, V := 0$
02. while $(A \neq B)$ do
03.   if ($A$ even) then
04.     $A := A/2$;
05.     if ($U$ even) then $U := U/2$ else $U := (U + M)/2$;
06.   else if ($B$ even) then
07.     $B := B/2$;
08.     if ($V$ even) then $V := V/2$ else $V := (V + M)/2$;
09.   else if $(A > B)$ then
10.     $A := (A - B)/2$;
11.     $U := (U - V)$;
12.     if $(U < 0)$ then $U := (U + M)$;
13.     if ($U$ even) then $U := U/2$ else $U := (U + M)/2$;
14.   else
15.     $B := (B - A)/2$;
16.     $V := (V - U)$;
17.     if $(V < 0)$ then $V := (V + M)$;
18.     if ($V$ even) then $V := V/2$ else $V := (V + M)/2$;
19. end while
20. return $U$;
```

18.5 Basic Division Architecture

As can be seen from algorithm 1, the divider makes decisions based on the parity and magnitude comparisons of m-bit registers A and B. To determine the parity of a number, only the Least Significant Bit (LSB) needs be examined (0 implies *even*, 1 implies *odd*). However, true magnitude comparisons can only be achieved through full m-bit subtractions, and thus introduce significant delay before decisions regarding the next computation can be made.

The first design presented here uses m-bit carry propagation adders to perform the additions/subtractions. At each iteration of the while loop in algorithm 1, U_{new} is assigned one of the following values, depending on the parity and relative magnitude of A and B: U, $\frac{U}{2}$, $\frac{(U+M)}{2}$, $\frac{(U-V)}{2}$, $\frac{(U-V+M)}{2}$ or $\frac{(U-V+2M)}{2}$.

The basis for the proposed designs is to calculate all 6 possible values concurrently, and use multiplexors to select the final value of U_{new}. This eliminates the time required to perform a full magnitude comparison of A and B before calculation of U and V can even commence. These architectures are illustrated in Fig. 18.3 and Fig. 18.4.

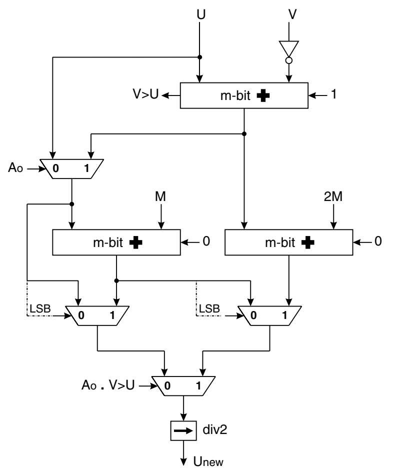

Figure 18.3. Determination of U_{new}.

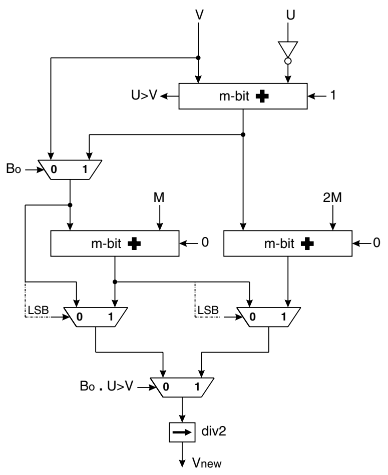

Figure 18.4. Determination of V_{new}.

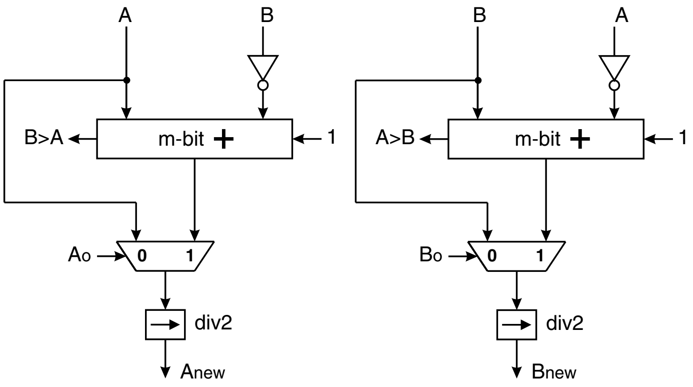

Figure 18.5. Determination of A_{new} and B_{new}.

The architecture for the determination of A_{new} and B_{new} is simpler as illustrated in Fig. 18.5. The U, V, A and B registers are also controlled by the parity and relative magnitudes of A and B, and are not clocked on every cycle.

18.6 Proposed Carry-Select Division Architecture

The clock speed is dependent on the bitlength of the divider since the carry chain of the adders contributes significantly to the overall critical path of the design. When the carry chain length exceeds the column height of the FPGA, the carry must be routed from the top of the column to the bottom of the next. This causes a significant decrease in the overall clock speed. The carry-select

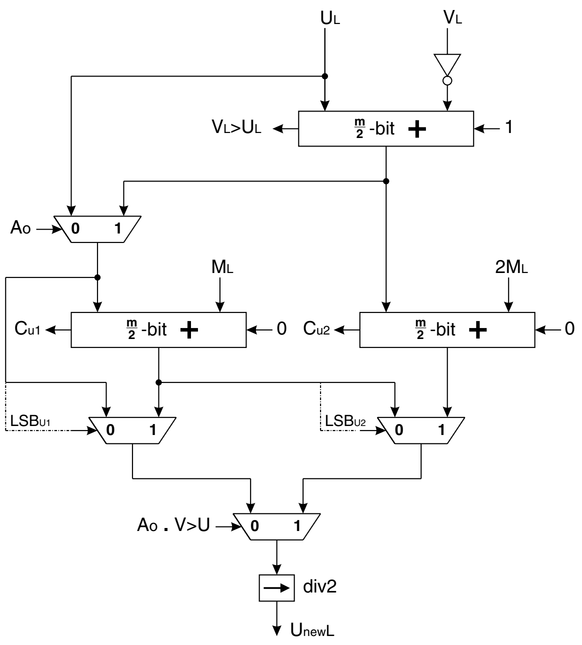

Figure 18.6. Carry-Select Architecture: Determination of U_{newL}.

design proposed here halves this adder carry chain at the expense of extra adders, multiplexors and control, but in doing so improves the performance of the divider.

This architecture is similar to a carry-select inverter design proposed in [Daly et al., 2003]. The values of $(U - V)$ and $(A - B)$ are determined by splitting the four m-bit registers U, V, A and B into eight $(\frac{m}{2})$-bit registers U_L, U_H, V_L, V_H, A_L, A_H, B_L and B_H. The values of $(U - V)_L$ and $(U - V)_H$ are then determined concurrently to produce $(U - V)$ as illustrated in Fig. 18.6 and Fig. 18.7.

When calculating $(U - V)$, if the least significant $(\frac{m}{2})$ bits of V are greater than the $(\frac{m}{2})$ least significant bits of U (i.e. $V_L > U_L$), then one extra bit must be "borrowed" from U_H to correctly determine the value of $(U - V)_H$. Therefore, the value of $(U - V)_H$ will actually be $(U_H - V_H - 1)$. It is observed that $(U_H - V_H - 1)$ is equivalent to $(\overline{V_H - U_H})$ in two's complement representation. Since the value of $(V_H - U_H)$ has been calculated in the determination of V_{new}, only a bitwise inverter is needed to produce $(U - V)_H$ as seen in Fig. 18.7. However, it is also possible that a carry from the addition of M_L or $2M_L$ will affect the final value of U_{newH}. Therefore carries of -1, 0 and 1 must be accounted for in the determination of U_{newH}. To allow for this, an extra $(\frac{m}{2})$-bit adder with a carry-in of 1 is required to calculate $(U_H - V_H + M_H + 1)$.

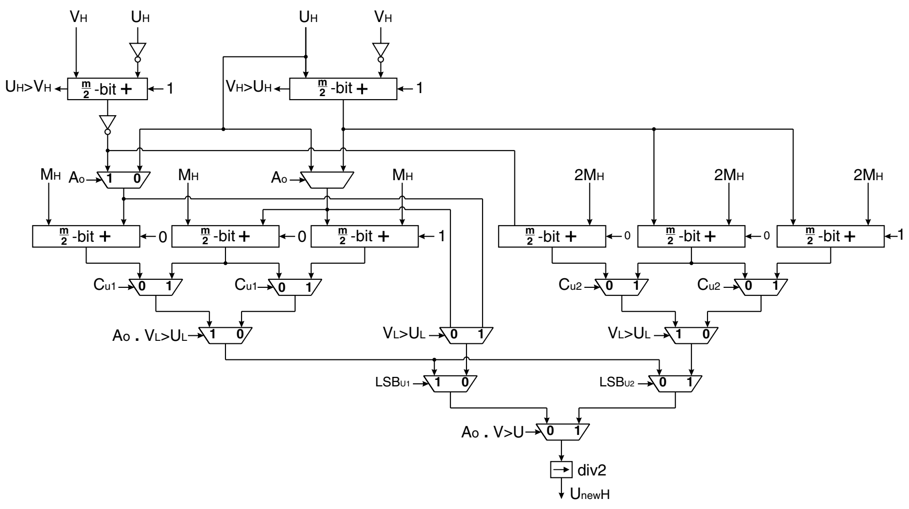

Figure 18.7. Carry-Select Architecture: Determination of U_{newH}.

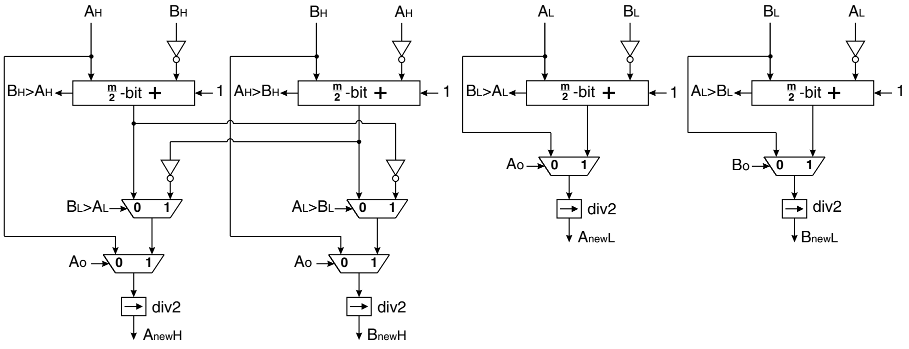

Figure 18.8. Carry-Select Architecture: Determination of A_{new} and B_{new}.

The determination of V_{new} is similar to that of U_{new}. The values of A_{new} and B_{new} are determined as illustrated in Fig. 18.8.

18.7 Results

Speed and area results for the two divider architectures are given in Tables 18.1 and 18.2. In each table the area and speed of each design are listed for 32, 64, 128 and 256-bit dividers, indicating the percentage increase in area and speed of the carry-select architecture over the basic architecture.

Table 18.1 lists the results targeted to the Xilinx VirtexE xcv2000e-6bg560 FPGA. This device has a maximum unbroken carry chain length of 160 bits (due to 80 Configurable Logic Blocks per column).

Table 18.1. Area and speed results for the two designs on xcv2000e-6bg560

Size	Design	Area (Slices)	% of xcv2000e	Area Increase	Max. Freq. (MHz)	Speed Increase
32-bit	*Basic*	561	2.9		63.16	
	Carry-Select	724	3.8	29.1%	57.25	−9.4%
64-bit	*Basic*	1,057	5.5		47.42	
	Carry-Select	1,351	7.0	27.8%	46.48	−2.0%
128-bit	*Basic*	1,992	10.4		30.90	
	Carry-Select	2,718	14.2	36.4%	35.18	13.9%
256-bit	*Basic*	4,353	22.7		19.85	
	Carry-Select	5,560	28.9	27.7%	26.23	32.1%

Table 18.2. Area and speed results for the two designs on xc2v1500-6bg575

Size	Design	Area (Slices)	% of xc2v1500	Area Increase	Max. Freq. (MHz)	Speed Increase
32-bit	*Basic*	551	7.2		103.39	
	Carry-Select	709	9.2	28.7%	91.62	−11.4%
64-bit	*Basic*	1,102	14.3		86.45	
	Carry-Select	1,398	18.2	26.9%	80.35	− 7.1%
128-bit	*Basic*	2,003	26.1		58.01	
	Carry-Select	2,622	34.1	30.9%	61.38	5.8%
256-bit	*Basic*	4,163	54.2		35.84	
	Carry-Select	5,268	68.6	26.5%	50.68	41.4%

Table 18.2 gives results targeted to the Xilinx Virtex2 xc2v1500-6bg575 FPGA. This device has a maximum unbroken carry chain length of 192 bits (due to 96 CLB's per column).

VHDL synthesis and place and route were performed on Xilinx ISE 5.1. The results are post place and route with a top level architecture to load the data in 32-bit words.

For 32-bit and 64-bit designs, the increased delay due to the extra level of multiplexors and control masks the reduction in the critical carry path of the adders. The delay introduced by the multiplexors is approximately 60 times that of a 1-bit carry propagation. This results in a decrease in speed for these designs.

For the 128-bit designs, the reduction of the carry chain from 128 bits to 64 bits in the carry-select design gives a slight speed improvement over the basic design. However, once the maximum carry chain length of the device has been exceeded, the basic architecture suffers a greater degradation in performance due to the routing of the carry from the top of one column to the bottom of the next. The 256-bit results show a much greater increase in speed for the carry-select design where the carry chain remains unbroken.

In all cases, the increase in area remains reasonably constant at approximately 30%.

Comparing these results to the inversion architecture results presented [Daly et al., 2003], it is observed that both inverter and divider have comparable operating clock frequencies.

The divider requires half the number of clock cycles to perform the operation, however it requires significantly more area. It is estimated that using this new architecture, division can be performed twice as fast as the alternative invert and multiply architecture.

18.8 Conclusions

Modular division is an important operation in elliptic curve cryptography. In this paper, two new FPGA architectures, based on a recently published division algorithm [Shantz, 2001] have been presented and implemented. The basic design computes all possible outcomes from each iteration and uses multiplexors to select the correct answer. This avoids the necessity to await the outcome of a full m-bit magnitude comparison before computation can begin. The second, carry-select divider design splits the critical carry chain into two, and again performs all calculations before the magnitude comparison has been completed. The 256-bit carry-select divider achieved a 40% increase in speed over the basic design at the cost of a 30% increase in area. The operating speed of the proposed divider is comparable to that of previously presented inverters on similar devices, but needs only half the number of clock cycles. Since division by inversion followed by multiplication requires many extra clock cycles, and hence more power, the proposed carry-select divider architecture seems better suited for implementation in an ECC processor.

Acknowledgments

This work is funded by the research innovation fund from Enterprise Ireland.

References

Blake, I., Seroussi, G., and Smart, N. (2000). *"Elliptic Curves in Cryptography"*. London Mathematical Society Lecture Note Series 265. Cambridge University Press.

Daly, A. and Marnane, W. (2002). "Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic". 10th Intl Symposium on FPGA (FPGA 2002), pages 40–49.

Daly, A., Marnane, W., and Popovici, E. (2003). “Fast Modular Inversion in the Montgomery Domain on Reconfigurable Logic”. *Irish Signals and Systems Conference ISSC2003*, pages 362–367.

Ernst, M., Jung, M., Madlener, F., Huss, S., and Blümel, R. (2002). “A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2^n)”. *Cryptographic Hardware and Embedded Systems—CHES 2002*, (LNCS 2523):381–398.

Gutub, A., Tenca, A. F., Savas, E., and Koc, C. K. (2002). “Scalable and unified hardware to compute Montgomery inverse in GF(p) and GF(2^n)”. *Cryptographic Hardware and Embedded Systems - CHES 2002*, (LNCS 2523):484–499.

IEEE (2000). IEEE 1363/D13 Standard Specifications for Public Key Cryptography.

Kaliski, B. S. (1995). “The Montgomery Inverse and it’s applications”. *IEEE Trans. on Computers*, 44(8):1064–1065.

Kerins, T., Popovici, E., Marnane, W., and Fitzpatrick, P. (2002). “Fully Parameterizable Elliptic Curve Cryptography Processor over GF(2^m)”. 12th *Intl Conference on Field-Programmable Logic and Applications FPL2002*, pages 750–759.

Koblitz, N. (1987). “Elliptic Curve Cryptosystems”. *Math Comp*, 48:203–209.

Miller, V. S. (1985). “Use of Elliptic Curves in Cryptography”. *Advances in Cryptography Crypto’85*, (218):417–426.

Orlando, G. and Paar, C. (2001). “A Scalable GF(p) Elliptic Curve Processor Architecture for Programmable Hardware”. *Cryptographic Hardware and Embedded Systems - CHES 2001*, (LNCS 2162):348–363.

Savas, E. and Koc, C. K. (2000). “The Montgomery Modular Inverse—Revisited”. *IEEE Trans. on Computers*, 49(7):763–766.

Shantz, S. C. (2001). “From Euclid’s GCD to Montgomery Multiplication to the Great Divide”. Technical Report TR-2001-95, Sun Microsystems Laboratories.