Dual-Field Multiplier Architecture for Cryptographic Applications

E. Savaş
Sabanci University
Istanbul, Turkey TR-34956

A. F. Tenca and Ç. K. Koç
Oregon State University
Corvallis, OR 97331

Abstract

The multiplication operation in finite fields $GF(p)$ and $GF(2^n)$ is the most often used and time-consuming operation in the harware and software realizations of public-key cryptographic systems, particularly elliptic curve cryptography. We propose a new hardware architecture for fast and efficient execution of the multiplication operation in this paper. The proposed architecture is scalable, i.e., can handle operands of any size; only limited by input/output and scratch space size, not by computational unit. It can also be configured to fit the available chip area for the desired performance. Our proposed architecture computes multiplication faster in $GF(2^n)$ than $GF(p)$, which conforms with premise of $GF(2^n)$ for hardware realizations.

I. Introduction

One of the motivations for fast and area efficient hardware solutions for multiplications in finite fields $GF(p)$ and $GF(2^n)$ comes from the fact that they are the most time-consuming operations in cryptographic applications such as the decipherment operation of the RSA algorithm [1], the Diffie-Hellman key exchange algorithm [2], the Government Digital Signature Standard [3] and also recently elliptic curve cryptography [4].

In this paper, following the design principles introduced in [7], we present a novel scalable multiplier architecture that is unified in the sense that multiplication for both $GF(p)$ and $GF(2^n)$ is performed in the same datapath. Furthermore, the novelty presented here is that the multiplier works with radix-4 for $GF(2^n)$ and radix-2 for $GF(p)$. Therefore, the architecture is referred as dual-radix. We will discuss the effects and advantages of these techniques on the chip area, signal propagation time and the clock cycle count to complete a multiplication operation.

A. Previous Work

The multiplier architecture presented here is based on the Montgomery multiplication algorithm which is originally proposed as an efficient method for doing multiplication operation in $GF(p)$ [5]. The algorithm replaces division operation with simple shifts, which are particularly suitable for implementation in hardware as well as in software on general-purpose computers. Therefore, the Montgomery multiplication algorithm generally allows to design a hardware unit with shorter signal propagation time (higher maximum clock frequency) besides taking advantage of certain design optimizations such as systolic array [6] and pipeline organizations [7].

In [8], it is also shown that Montgomery multiplication might be very efficient in $GF(2^n)$, when polynomial basis is used. Since the steps of the Montgomery multiplication algorithm for both fields are almost identical, it is possible to design a unified architecture. Feasibility and advantage of designing such a unified multiplier architecture for elliptic curve cryptography have been extensively discussed in [7], [9].

Various hardware implementations of the Montgomery multiplication algorithm for limited precision operands were proposed in [10]. Implementations utilizing high-radix modular multipliers have also been proposed in [11]. Aspects of using high-radix representation have been discussed in [12]. Even though very high-radix designs have certain complications in hardware, moderate radix values offer faster alternatives to simple radix-2 multiplier designs.

The original unified multiplier in [7] uses radix-2 design and offers an equal performance for both $GF(p)$ and $GF(2^n)$ of same precision. For this very reason, however, the original design is not optimized since it does not take the advantage of using $GF(2^n)$, which is, in general, more efficient than $GF(p)$ in hardware implementations. Our first observation is that this situation can be remedied by putting to use the part of the circuitry which is underutilized in $GF(2^n)$ mode. This allows us to run the multiplier module in higher radix values for $GF(2^n)$ than those for $GF(p)$ without significantly increasing the design complexity.

B. Montgomery Multiplication Algorithm

In [5], Montgomery described a modular multiplication method which proved to be very efficient in both hardware and software implementations. An obvious advantage of the method is the fact that it replaces division operations with simple shift operations. The method adds multiples of the modulus rather than subtracting it from the partial result. Refer to [5]. for detailed explanation of the algorithm.

Given two integers a and b, and a prime modulus p, the Montgomery multiplication algorithm computes $\bar{c} = MonMult(a,b) = a \cdot b \cdot R^{-1} \pmod{p}$ where $R = 2^n$ and $a, b < p < R$ and p is an n-bit prime number. The Montgomery multiplication does not directly compute $c = a \cdot b \pmod{p}$, therefore certain transformation operations must be applied to the operands a and b before the multiplication and to the intermediate result $\bar{c}$ in order to obtain the final result c. T

The Montgomery multiplication algorithm with radix-2^k for $GF(p)$ can be given as in the following:

Algorithm A
Input: $a, b \in [1, p-1]$, p, and m
Output$c \in [1, p-1]$
1: $c := 0$
2: for $i = 0$ to $m-1$
3: $\quad q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$

4: $c := (c + a_i \cdot b + q \cdot p)/2^k$

where $p_0' = 2^k - p_0^{-1} \pmod{2^k}$. In the algorithm, the multiplier a is written with base (radix-2^k) and digits a_i so that $a = \sum_{i=0}^{m-1} a_i \cdot 2^{k \cdot i}$, where m is the number of digits in a and $m = \lceil n/k \rceil$. In Step 4, the multiplicand b, the modulus p, and the partial result c enter the computations as full-precision integers. However, in our implementation we will treat b, p, and c as multi-word integers in order to design a scalable multiplier and in each clock cycle one word of these values will be processed. One may also consider this representation as writing the multiplicand, the modulus and the partial result with digits $b^{(j)}$, $p^{(j)}$, and $c^{(j)}$ of w bits, so that $b = \sum_{j=0}^{e-1} b^{(j)} \cdot 2^{w \cdot j}$, $p = \sum_{j=0}^{e-1} p^{(j)} \cdot 2^{w \cdot j}$, and $c = \sum_{j=0}^{e-1} c^{(j)} \cdot 2^{w \cdot j}$ where $e = \lceil n/w \rceil$. Note that the base-2^w used to represent b, p, and c in Step 4 is different from the radix-2^k used to represent the multiplier a in Step 3. Note also that q, c_0, b_0, and p_0' are all k-bit integers.

In order to avoid a possible confusion due to the usage of two different bases, we elect to refer the digits of b, p and c as words when implementing Step 4, and use the term *digit* exclusively for the multiplier a, and for b_0, p_0', and c_0 in Step 3 when they are in the same equation with the digits of a. Digits can be easily distinguished by the subscript notation (e.g. a_i or b_0) from superscript notation of word (e.g. $b^{(j)}$). We will also use the notation $x_{i,j}$ to denote the jth bit in the ith digit of x.

In addition, the radix of the multiplier architecture is determined by the base used to represent the multiplier a.

The Montgomery multiplication algorithm for $GF(2^n)$ is given below:

Algorithm B

Input: $a(x), b(x)$, $p(x)$, and m
Output: $c(x)$
1: $c(x) := 0$
2: for $i = 0$ to $m-1$
3: $q(x) := (c_0(x) + a_i(x) \cdot b_0(x)) \cdot p_0'(x) \pmod{x^k}$
4: $c(x) := (c(x) + a_i(x) \cdot c(x) + q(x) \cdot p(x))/x^k$

where $p_0'(x) = p_0^{-1}(x) \pmod{x^k}$. The two algorithms are almost identical except that the addition operation in $GF(p)$ becomes a bitwise modulo-2 addition in $GF(2^n)$. In Algorithm A, there must be an extra reduction step at the end to reduce the result into the desired range if it is greater than the modulus. On the other hand, this step is not essential part of the algorithm and there are simple conditions that can be added to the algorithm in order to eliminate it [13], hence we intentionally exclude it from the algorithm definitions.

From this point on, we will only use the notation introduced in Algorithm A for both $GF(p)$ and $GF(2^n)$ and leave polynomial notation completely out of our representation of field elements in $GF(2^n)$. Operations will be deduced from the mode ($GF(p)$ or $GF(2^n)$) in which the module is operated. The elements of both fields are represented identically in the digital systems.

C. Precomputation in Montgomery Multiplication Algorithm

The unified multiplier architecture introduced in the next section utilizes a precomputation technique in order to decrease the critical path delay of the original unified multiplier in [7]. Note that Step 4 of the Algorithm A computes

$$c := (c_0 + a_i \cdot b + q \cdot p)/2^k.$$

Depending on the radix value chosen, the LSDs of the operands, a_i, b_0, and c_0 will determine which one of the values in $\{0, b, p, b+p, 2p, 2b, 2b+2p, \ldots\}$ is added to the partial result c. If one precomputes and stores the value of $b+p$, the calculations in Step 4 can be significantly simplified.

There are three implications of the precomputation technique. First, the fact that an adder must be available to perform the precomputation potentially leads to an increase in the chip area. However, we show that such an adder is already an integral part of our design and the precomputation will be done without any extra overhead in this sense. Second, the precomputed value must be stored. This will imply an increase in the register space. And finally, there must be a so-called selection logic to select which multiples of b and p must participate in the addition in Step 4. The selection logic will be naturally on the critical path and can potentially result in both an increase in the chip area and critical path delay. On the other hand, the precomputation technique also simplifies the design since Step 4 can be performed with only one addition, once the selection logic generates its output. We will provide implementation results to expose the effects of the precomputation technique in the multiplier design.

II. Radix-(2,4) Multiplier Architecture

In this section, we present a unified and scalable multiplier architecture which operates in radix-2 in $GF(p)$ mode and in radix-4 in $GF(2^n)$ mode. and the architecture is called radix-(2,4).

A. Processing Unit

In this section, we explain the design details of the processing unit (PU) which is basically responsible for performing Step 3 and Step 4 of Algorithm A.

Since the multiplier uses radix-2 for $GF(p)$, the least-significant bits (LSB) of the operand digits, a_i, b_0, and c_0 will determine which one of the values in $\{0, b, p, b+p\}$ is added to the partial result c. In the case of $GF(2^n)$, multiplication is performed in radix-4. Therefore, the LSDs (least significant digits) of b, p, and c and of the current digit of a are in order to determine q. The LSB of p is always 1, then only $p_{0,1}$, the second least significant bit of the modulus, is included in the computations. Consequently, $a_{i,1}, a_{i,0}, b_{0,1}, b_{0,0}, c_{0,1}, c_{0,0}$ and $p_{0,1}$ determine one of the following values to be added to the partial result: $\{0, b, p, b+p, x \cdot b, x \cdot p, x \cdot (b+p)\}$.

In Figure 1, the architecture of the processing unit (PU) used in the dual-radix multiplier is illustrated. The local control logic in Figure 1 contains the selection logic which generates the signals, m_{00}, m_{01}, m_{10},

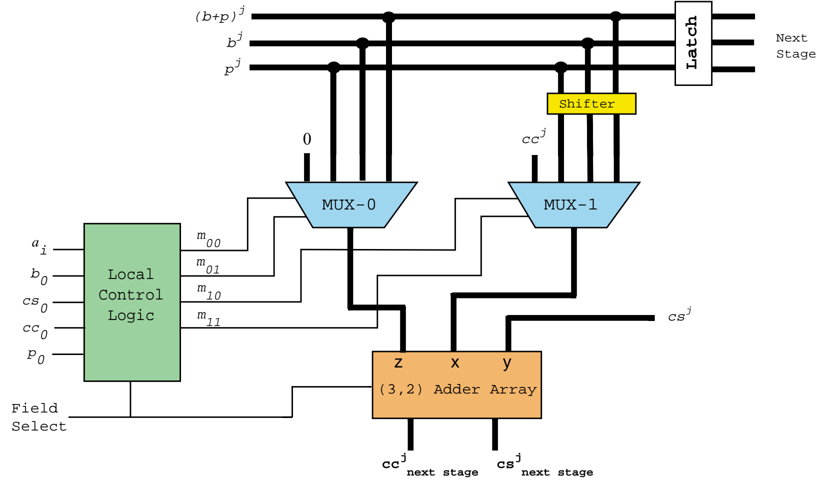

Fig. 1. Processing unit of dual-radix architecture with radix-2 for $GF(p)$ and radix-4 for $GF(2^n)$

and m_{11}, to determine which multiples of b and p will be in the calculations. cc_0 and cs_0 in Figure 1 are the least significant digits of carry and sum part of the partial result c.

The dual-radix architecture consists of one or more processing units (PU), identical to the one shown in Figure 1, organized in a pipeline. Each PU takes a digit (k-bits) from the multiplier a, the size of which depends on the radix and the mode (finite field), and operates on the words of b, $b+p$, c and p successively starting from the least significant word. Starting from the second cycle it generates one word of partial result each cycle which is communicated to the next PU. After $e+1$ clock cycles, where e is the number of words in the modulus (i.e. $e = \lceil n/w \rceil$), a PU finishes its portion of work and becomes free for further computation. One can refer to [7] for more information about the pipeline organization.

A redundant representation (Carry-Save) is used for the partial result in the architecture. Thus, for the partial result we can write $c = cc + cs$, where cc and cs stand for the carry and sum part of the partial result. Redundant format necessitates an extra addition operation to transform the final result into nonredundant format at the end of the calculations. The transformation operation is simply performed by a carry propagate adder (e.g. carry look-ahead adder) which is also capable of doing modulo-2 addition operation in $GF(2^n)$-mode. The existence of an adder is also useful for performing the precomputation of $b+p$, which is used during multiplication.

B. (3, 2) Adder Array

An n-bit (3, 2) adder array shown in Figure 1 consists of two parts: single-bit dual-field adders (DFA) and shift-and-alignment layer as demonstrated in Figure 2. When used in $GF(p)$-mode, the DFA simply becomes a Carry-Save adder. A DFA cell is basically a full-adder capable of doing addition with or without carry. It has an input called $FSEL$ that enables this functionality. Our implementation results demonstrated that this additional functionality is obtained almost without any cost.

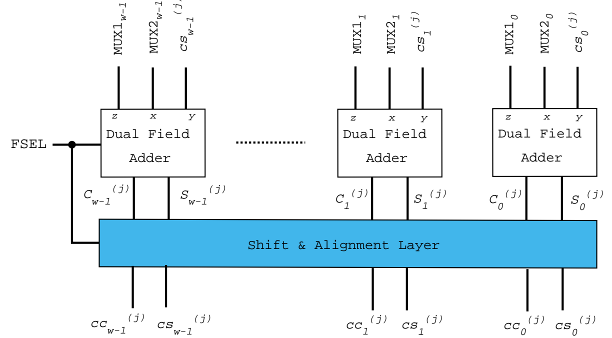

Fig. 2. Dual Field Adder Array for radix-(2,4) Unified Multiplier

C. Selection Circuitry

As stated previously, the selection logic for radix-(2,4) multiplier, which is shown in Figure 3, determines which of the inputs of MUX-0 and MUX-1 in Figure 1 are to be added in $(3,2)$ adder array, which in turn calculates $c := c + a_i \cdot b + q \cdot p$.

In $GF(p)$-mode the multiplier uses radix-2, hence m_{00} and m_{01} must be calculated while m_{10} and m_{11} are forced to be 0 since input 0 of MUX-1 is always selected in this mode. We can use the following formulae to express the control inputs of MUX-0.

$$m_{00} = a_{i,0}$$

$$m_{01} = q_0 = (cs_{0,0} \oplus cc_{0,0} \oplus a_{i,0} \cdot b_{0,0})$$

where $\oplus$ stands for modulo-2 addition, $a_{i,j}$ denotes jth bit of the digit a_i and q_j is the jth bit of q, and $cs_{i,j}$ and $cc_{i,j}$ are the sum and carry bits of the partial result, respectively.

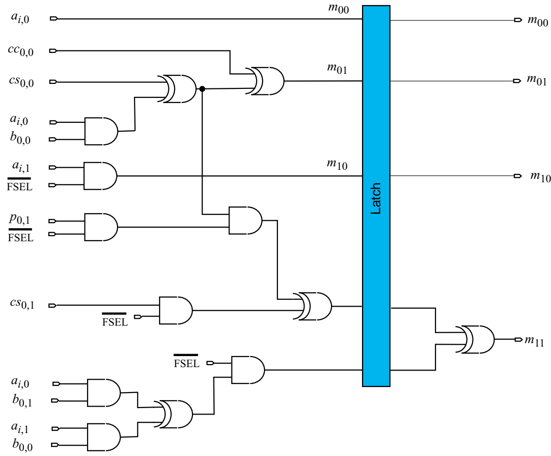

Fig. 3. Selection logic for radix-(2,4) multiplier

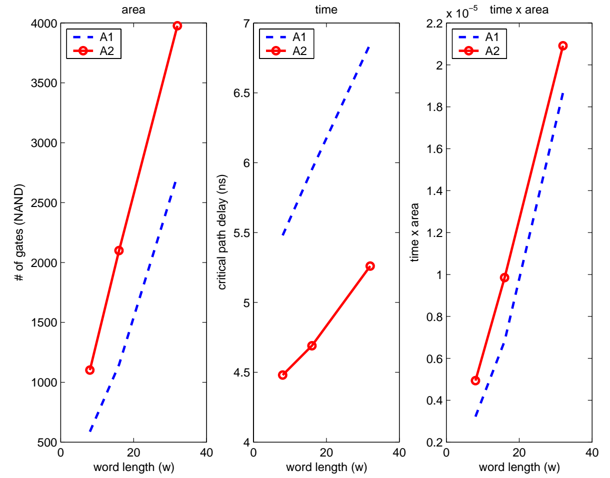

Fig. 4. Implementation results: Critical path delay and area

On the other hand, the multiplier computes with radix-4 in $GF(2^n)$-mode. Thus, the select inputs of MUX-1 must also be calculated. For this, we use the formulae

$$m_{10} = a_{i,1} \cdot \overline{FSEL}$$

$$m_{11} = q_1 = [(cs_{0,1} \oplus a_{i,0} \cdot b_{0,1} \oplus a_{i,1} \cdot b_{0,0}) \oplus (cs_{0,0} \oplus a_{i,0} \cdot b_{0,0}) \cdot p_{0,1}] \cdot \overline{FSEL}$$

Note that the first input of MUX-1, cc is always zero in this mode since redundant form is also used for partial result and the carry part of it is forced to be zero.

III. Implementation results

We implemented processing units of two different multiplier architectures: **(A1)** the original unified multiplier in [7], and **(A2)** radix-(2,4) multiplier. We used VHDL to implement two architectures and synthesized the resulting code using Mentor Graphics tools for an ASIC technology of $0.5\mu m$ AMI CMOS (ADK library [14]).

Figure 4 demonstrates the area and time delay of two different PU designs, using different word sizes. Area consumption is always given in terms of 2-input **NAND** gates. Due to the highly modular nature of the design, the critical path of a PU determines the maximum clock frequency that can be applied to the whole multiplier.

As can easily be observed from Figure 4, there is an increase in area of the new architecture. There are two basic reasons for this increase: (1) having an extra interstage register for passing the precomputed value, $b + p$, to the next stage, (2) selection logic. The selection logic becomes more complicated due to what may be appropriately called as a *look-ahead* technique which processes the least-significant bits of the operands. The fact that two least significant bits of some operands are needed in the look-ahead technique partially explains the further increase in the area. More complicated shift-and-alignment layer is another reason for larger area usage in the dual-radix design. Note also that, the relative increase in area becomes less significant as the word size also increases. This can also be explained by the fact that the area of selection logic is independent of word size. When $w = 32$, the area consumed by the selection logic becomes less significant. For example, increase in area in the dual-field multiplier (**A2**), 45% when $w = 32$. The use of the precomputation technique in the architecture **A2** improves the critical path delay by 18% to 23%.

The performance of the two multipliers in terms of clock cycle count to perform a multiplication is determined, to a large extent, by the number of PUs (t) and the word size (w), which is subject to the limitations on the silicon area available. Therefore, the relative increase in the area of a PU may be misleading in evaluating the overall performance of the new architectures. Two architectures utilize many PUs organized in a pipeline. To provide more insight in the overall effect of the new architecture on the area and time, we investigated the time to compute multiplication for a precision range of cryptographic interest given a limited area. Figure 5 demonstrates the results for multiplier configuration in $GF(p)$-mode with approximately $30,000$ gates. We basically designed the multipliers for each architecture by putting as many PUs as possible.

In this configuration, the new architecture, **A2**, offer a significant speedup in time performance over the original architecture **A1** for the range of $[160, \sim 500]$. Beyond the precision of 500 bits, higher area requirements of new architectures will have a negative impact on the performance. For the same area the new architecture, **A2** is by 13% to 35%. Note that the maximum speedup in the new architectures, exceeds the maximum speedup provided by a single PU. This is due to the fact that having more PUs not always improves the performance, hence may result in a slight degradation for some bit lengths. The dual-

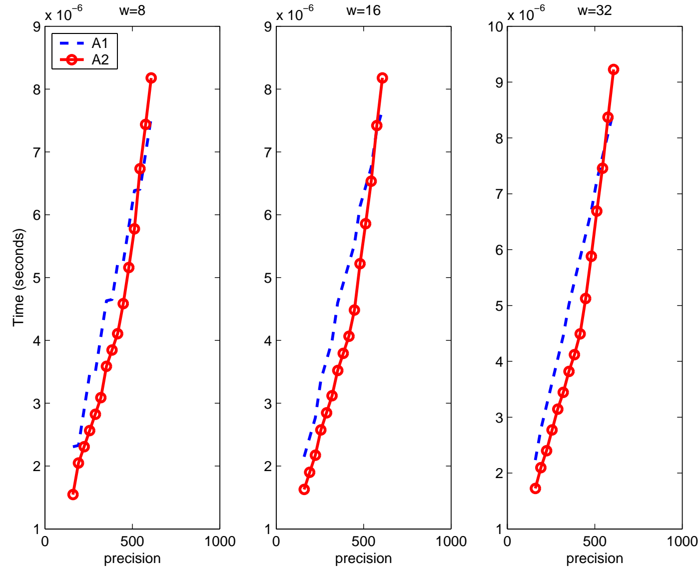

Fig. 5. Multiplication Timings (in μs) for an area of 30.000 gates with $w = 8, 16$, and 32 in $GF(p)$-mode

radix architecture offers a significant speedup over **A1** in $GF(2^n)$-mode. It outperforms **A1** by 56% to 67% in this mode.

IV. Summary and Conclusions

Using the design methodology proposed in [7], we presented a new unified multiplier architecture called dual-radix architecture for binary extension and prime fields. The architecture utilizes a precomputation technique and improves critical path delay significantly. The cost of implementing the precomputation technique in hardware in terms of area is studied and it has been concluded that the overall impact is insignificant for a large range of precision. The dual-radix architecture also facilitates faster computation of multiplication in $GF(2^n)$-mode than $GF(p)$-mode. The area and speed characteristics of the dual-radix architecture is also extensively investigated and its performance in terms of area and time is compared against single-radix, unified multiplier architecture. At the expense of using extra resources, which proved to have a very limited impact on the silicon area under certain circumstances, it provides significant improvement in critical path delay compared to the original unified design in both $GF(p)$ and $GF(2^n)$-modes. Furthermore, it provides a superior performance in $GF(2^n)$-mode.

References

[1] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.

[2] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[3] National Institute for Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-2, January 2000.

[4] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[5] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[6] Colin D. Walter. An improved linear systolic array for fast modular exponentiation. *IEE Proceedings - Computers and Digital Techniques*, 147(5):323–328, Sept. 2000.

[7] E. Savaş, A. F. Tenca, and Ç. K. Koç. A scalable and unified multiplier architecture for finite fields GF(p) and GF(2^m). In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000*, Lecture Notes in Computer Science No. 1965, pages 281–296. Springer, Berlin, Germany, 2000.

[8] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

[9] Johann Grossschadl. A bit-serial multiplier architecture for finite fields GF(p) and GF(2^k). In *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 2162, pages 202–219. Springer-Verlag, Berlin, 2001.

[10] A. Bernal and A. Guyot. Design of a modular multiplier based on Montgomery's algorithm. In *13th Conference on Design of Circuits and Integrated Systems*, pages 680–685, Madrid, Spain, November 17–20 1998.

[11] P. Kornerup. High-radix modular multiplication for cryptosystems. In E. Swartzlander, Jr., M. J. Irwin, and G. Jullien, editors, *Proceedings, 11th Symposium on Computer Arithmetic*, pages 277–283, Windsor, Ontario, June 29 – July 2 1993. IEEE Computer Society Press, Los Alamitos, CA.

[12] C. D. Walter. Space/Time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, 46(2):139–141, February 1997.

[13] Colin D. Walter. Montgomery exponentitation needs no final subtractions. *Electronic Letters*, 35(21):1831–1832, October 1999.

[14] ASIC design kit. Mentor Graphics Co.