AREA EFFICIENT $GF(p)$ ARCHITECTURES FOR $GF(p^m)$ MULTIPLIERS

Jorge Guajardo Thomas Wollinger Christof Paar

Communication Security Group (COSY), Ruhr-Universität Bochum, 44780 Bochum, Germany.
Email: {guajardo,wollinger,cpaar}@crypto.ruhr-uni-bochum.de

Presented at the 45th IEEE International Midwest Symposium on Circuits and Systems - MWSCAS 2002, Tulsa, Oklahoma, August 4-7, 2002

ABSTRACT

This contribution describes new $GF(p)$ multipliers, for $p > 2$, specially suited for $GF(p^m)$ multiplication. We construct truth tables whose inputs are the bits of the multiplicand and multiplier and whose output are the bits that represent the modular product. However, contrary to previous approaches, we don't represent the elements of $GF(p)$ in the normal binary positional system. Rather, we choose a representation which minimizes the resulting Boolean function. We obtain improvements of upto 35% in area when compared to previous approaches for small odd prime £elds. We report transistor counts for all multipliers with $p < 2^5$ which we obtained through the SIS Sequential Circuit Synthesis program.

1. INTRODUCTION

Modulo p multipliers have received a lot of attention in the research literature as they are fundamental building blocks in DSP applications which make use of the Residue Number System (RNS). Also, public-key cryptosystem applications based on the Discrete Logarithm (DL) problem (for example the Dif£e-Hellman protocol), on the Integer Factorization problem (e.g. RSA), and, more recently, applications based on the DL problem in elliptic curves, all make use of large $GF(p)$ or $GF(2^k)$ multipliers (160–1024 bit long operands). Hardware solutions have always focused on £elds of characteristic 2 due to the straight forward manner in which elements of $GF(2)$ can be represented.

In recent years, $GF(p^m)$ £elds, where p is odd and relatively small (between 2 and 64 bits), have gained interest in the research community. Bailey and Paar [1] introduced the concept of Optimal Extension Fields (OEFs) in the context of elliptic curve cryptography. OEFs are £elds $GF(p^m)$ where p is odd and both p and m are chosen to match the particular hardware used to perform the arithmetic, thus allowing for ef£cient £eld arithmetic. In [6, 9], $GF(p^m)$ £elds are proposed for cryptographic purposes where p is relatively small. In particular, [6] describes an implementation of ECDSA over £elds of characteristic 3 and 7. The author in [9] describes a method to implement elliptic curve cryptosystems over *small* £elds of odd characteristic, only considering $p < 24$ in the results section. More recently, Boneh and Franklin [3] introduced an identity-based encryption scheme based on the Weil and Tate pairings. These applications of the Weil and Tate pairings consider elliptic curves de£ned over £elds of characteristic 3. All these works have only considered *software*-based solutions.

Despite the research community's interest on systems based on small £elds of odd characteristic, to our knowledge, there are no contributions dealing with the implementation of such £elds in hardware or the more general topic of hardware architectures for OEFs. In this contribution we propose a method to design $GF(p)$ multipliers, for primes $2 < p < 2^5$, which are suitable for building $GF(p^m)$ multipliers.

1.1. Our Contributions

Hiasat [5] was the £rst to propose the design of modulo multipliers using the combinatorial logic approach by taking advantage of the fact that for any prime p there will always be $2^n - p$ *don't-care* positions in the truth table that de£nes the multiplier (where $n = \lceil \log_2 p \rceil$). In his paper, Hiasat only considers the *normal* approach to code the elements of $GF(p)$. For example, one considers the bit-strings $\{'000', '001', '010', '011', '100'\}$ to represent the numbers modulo 5, and the bit strings $\{'101', '110', '111'\}$ are simply used as *don't care* terms. We noticed, however, that one can choose a different representation. In other words, we could use the code words $\{'100', '101', '110', '111'\}$ to represent the set of integers {1,2,3,4}, and use the remaining *unused* bit-strings to represent the integer zero. Our results indicate that by carefully selecting the bit-strings representing the number zero, we obtain up to a 35% improvement in area over previous proposed architectures. The remaining of this contribution is organized as follows. Section 2 gives a survey of previous architectures proposed for the implementation of multiplication based on Residue Number Systems (RNS). Section 3 describes a new method to develop hardware architectures for modulo multipliers where the modulus is prime and small. We £nish our paper by analyzing our multiplier and comparing it to previously proposed architectures.

2. PREVIOUS WORK

RNS has been very popular as an implementation trick in DSP applications. An RNS is de£ned in terms of its moduli set, which consists of N relatively prime positive integers. The Chinese Remainder Theorem (CRT) allows one to map a large integer to a set of N residues belonging to small integer sets. Then, addition, subtraction, and multiplication are performed in parallel in each ring using modular arithmetic and, at the end, the result is converted back to the regular representation. The literature on VLSI architectures for RNS operations is extensive and can be classi£ed into four types: architectures based on memory or table look-ups, combinatorial architectures, mixed combinatorial/memory architectures, and adder-based architectures [2].

ROM-based designs are the simplest and are usually implemented via the use of table look-ups. The size of the table look-ups can be reduced using the fact that there exists an isomorphism between the multiplicative group of $GF(p)$ and the additive group modulo $p-1$ [8]. The fundamental problem with ROM-based designs is that they grow exponentially with the size of the input, thus making them ineffcient for large moduli.

Hiasat [5] introduced a new approach for designing modulo multipliers based on combinatorial logic. For a given prime, [5] constructs a truth table whose inputs are the bits of the multiplicand and the multiplier. The output bits are the product of the two inputs modulo the given prime. Given that one needs $n = \lceil \log_2 p \rceil$ bits to represent the numbers modulo any p, there are 2^n possible code words and only p of them are used, thus the remaining $2^n - p$ words can be used as *don't-cares* to minimize the number of minterms of each output bit. This design approach reduces by upto 50% the number of transistors required for small modulo multipliers ($p < 2^6$), when compared to ROM-based solutions, and it constitutes the basis for our new design methodology.

Various adder-based modulo multipliers have also been proposed in the literature [4, 10, 7]. Di Claudio et al. [4] introduced the pseudo-RNS representation. This new representation is similar in ¤avor to the Montgomery multiplication technique as it defnes an auxiliary modulus A relatively-prime to p. The technique allows building reprogrammable modulo multipliers, systolization, and simplifes the computation of DSP algorithms. Nevertheless, ROM-based solutions seem to be more effcient for small moduli $p < 2^6$ [4]. In [10] Soudris et al. present full-adder (FA) based architectures for RNS multiply-add operations which adopt the carry-save paradigm. The paper concludes that for moduli $p > 2^5$, FA based solutions outperform ROM ones. Finally, [7] introduces a new design which takes advantage of the non-occurring combinations of input bits to reduce certain 1-bit FAs to OR gates and, thus, reduce the overall area complexity of the multiplier. The multiplier outperforms in terms of area all previous designs. As a fnal remark, we would like to point out that in terms of time complexity, the designs in [4, 8] as well as ROM-based ones outperform the multiplier proposed in [7] for most prime moduli $p < 2^7$, however, the combined time/area product in [7] is always less than that of other designs.

2.1. Notation

In this section, we present the basic notation and definitions used in the sequel. Let $I = \{0, 1\}$ and $O = \{0, 1, -\}$, then a logic function f in t input variables $x_{t-1}, x_{t-2}, \ldots, x_1, x_0$ and s output variables $y_{s-1}, y_{s-2}, \ldots, y_1, y_0$ can be defned as:

$$f: \ I^t \rightarrow O^s$$

where $X = [x_{t-1}, x_{t-2}, \ldots, x_1, x_0] \in I^t$ is the input and $Y = [y_{s-1}, y_{s-2}, \ldots, y_1, y_0] \in O^s$ is the output. We noticed that in addition to the usual values of 0 and 1, the outputs y_i can also take on a *don't care* value $-$. Such functions are called incompletely specifed logic functions.

An element $A \in GF(p)$, will be represented as a binary string $[a_{n-1}, a_{n-2}, \ldots, a_1, a_0]$ of length $n = \lceil \log_2 p \rceil$. We point out that the binary encoding of A does not necessarily imply a positional number system. Whenever we imply the representation of A in radix-2 notation we write $(A)_2 = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)_2$ explicitly. We refer to the radix-2 representation of A as the *natural* or *normal* encoding of A interchangeably throughout the text.

For the purposes of this paper, multiplication in $GF(p)$, is an incompletely specifed logic function from I^{2n} to O^n. Here $X = [a_{n-1}, a_{n-2}, \ldots, a_1, a_0, b_{n-1}, b_{n-2}, \ldots, b_1, b_0] \in I^{2n}$ is the concatenation of the encodings of $A, B \in GF(p)$ and $C = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0] \in O^n$, with $C = A \cdot B \bmod p$. Finally, we represent the logical AND and OR operators, by $\wedge$, $\vee$, and we will use a bar over a binary variable to denote logical negation (i.e NOT$(a) = \overline{a}$).

3. $GF(p)$ ARITHMETIC FOR $P < 2^5$

$GF(p)$ multipliers are essential components used to build $GF(p^m)$ multipliers. The following two sections are devoted to describing the methodology used to design area optimized $GF(p)$ multipliers. The first part will concentrate exclusively on the $GF(3)$ case, which is of great interest as demonstrated by recent applications in the crypto community [6, 9, 3]. Our design methodology is generalized for odd prime fields, where $p < 2^5$. At the end of this section, we brie¤y consider the complexity of modulo-p adders, as they are also integral parts of a $GF(p^m)$ multiplier.

3.1. $GF(3)$ Multiplier

As mentioned before, [5] only considers the *normal* approach to encode elements of $GF(p)$. We noticed, however, that one can choose a different representation. We illustrate our approach by considering the case of $p = 3$.

Example 1. Let $p = 3$, then we have that $n = 2$ bits are required to represent any element of $GF(3)$. Using Hiasat's approach, one obtains the following Boolean equations to represent the product $C = (c_1, c_0)_2 = A \cdot B \bmod 3$ with $A = (a_1, a_0)_2$, $B = (b_1, b_0)_2$ and $A, B, C \in GF(3)$.

$$c_1 = (a_1 \wedge b_0) \vee (a_0 \wedge b_1), \quad c_0 = (a_0 \wedge b_0) \vee (a_1 \wedge b_1) \quad (1)$$

This method allows us to implement this boolean function with 4 AND gates and 2 OR gates. However, we can do better. Notice that the above equations were obtained with the natural encoding. If instead we encode the integers 1 and 2 with the binary strings $'10'$ and $'11'$, respectively, and allow the binary strings $\{'00', '01'\}$ to both represent the integer 0, we can represent modulo 3 multiplication as shown in Table 1. Applying logic minimization to Table 1,

A $[a_1, a_0]$	B $[b_1, b_0]$	C $[c_1, c_0]$	A $[a_1, a_0]$	B $[b_1, b_0]$	C $[c_1, c_0]$
00	00	0−	10	00	0−
00	01	0−	10	01	0−
00	10	0−	10	10	1 0
00	11	0−	10	11	1 1
01	00	0−	11	00	0−
01	01	0−	11	01	0−
01	10	0−	11	10	1 1
01	11	0−	11	11	1 0

Table 1: $C = A \cdot B \bmod 3$ truth table with 0 represented as $'00'$ or $'01'$, 1 as $'10'$, and 2 as $'11'$

one obtains the following boolean equations

$$c_1 = a_1 \wedge b_1, \quad c_0 = (\overline{a_0} \wedge \overline{b_0}) \vee (a_0 \wedge b_0) \quad (2)$$

Equation (2) can be realized with 2 NOT gates, 3 AND gates and 1 OR gate. This represents an improvement of about 30% with respect to [5] in the gate count if one does not take into account

the circuits required to convert to and from our modi£ed representation. In Section 3.4, we argue that for our particular application we can ignore such cost.

The optimized encoding of Example 1 was obtained by trying out all twelve possible encodings for $GF(3)$ elements in which we allow a redundant representation for the element 0 (i.e. one allows the element 0 to be represented by two different bit-strings[1]). In general, one has $\binom{2^n}{2^n-p+1}$ choices to encode the zero element of $GF(p)$ and $(p-1)!$ possible encodings for the non-zero elements. Multiplying these two numbers out one gets $\frac{2^n!}{(2^n-p+1)!}$ possible encodings.

Once an encoding has been chosen, one creates the corresponding truth table which is used as input to the well known Boolean minimization program ESPRESSO. We use the SIS program and its *script.algebraic* to £nd common terms in the Boolean functions obtained as output from ESPRESSO. One lesson we learned while performing this exhaustive search approach is that the encoding of the non-zero elements of $GF(3)$ does not matter for minimization purposes. Thus, for example, whether we encode the element 1 as $'10'$ and 2 as $'11'$ or viceversa, the same number of gates are required to implement the resulting Boolean functions. Assuming that the encoding of the non-zero elements of $GF(p)$ does not in¤uence the complexity of the minimized Boolean functions, we are left with only $\binom{2^n}{2^n-p+1}$ possibilities to encode the zero element. This means, for example, that we would have to perform 70 Boolean minimizations for $p = 5$ and 728 minimizations for $p = 11$. It is obvious from the start that such a methodology is only applicable for the smallest of primes. It is, thus, necessary to come up with ef£cient heuristics (or design criteria) that allow us to spot *good* encodings, in other words, encodings that minimize our Boolean equations. The next section introduces a set of rules that allows one to £nd *good encodings* without trying out all possible ones.

3.2. Modulo Multipliers for $p < 2^5$

Example 1 considered the particular case of $p = 3$ which is of the form $2^k + 1$. We notice that the encoding that produced the best results corresponded to representing an element $A \in GF(3) \setminus \{0\}$ using the *normal* binary representation of A' such that $A' = A+1$ and the element zero is encoded as $'00'$ or $'01'$. For $p = 5$, we found $A' = A + 4$ to be optimum for $A \in GF(5) \setminus \{0\}$ and we represent zero with the strings in the set $\{'000','001','010','011'\}$. A similar procedure can be applied to £nd a good encoding for the elements of $GF(17)$. This allows us to give our £rst heuristics.

Design Criterion 1 *Let $p = 2^k + 1$ be a prime. Then, we can decrease the area complexity of a combinatorial $GF(p)$ multiplier by encoding $A \in GF(p) \setminus \{0\}$ using the binary representation of A' with $A' = A + 2^n - 1$ and letting the remaining unused encodings to all represent the element zero. In addition, whenever A and/or B, in the multiplication $C = A \cdot B \bmod p$, assume any of the encodings of zero, the result C should be written as $'0\underbrace{-\cdots-}_{n\,don't\,cares}'$.*

[1]This is not to be confused with the technique of Redundant Number Representation used to limit carry propagation inherent in adder-based solutions for multipliers.

Intuitively, it is easy to see that for primes $p = 2^k+1$, the encoding of Criterion 1 allows one to detect the zero element by looking only at the £rst bit of the encoding of any $A \in GF(p)$. This is similar to the idea of diminished-1 representation so common in Number Theoretical Transform implementations. Unfortunately, the only primes $p = 2^k + 1 < 2^5$ are 3, 5, and 17. Thus, it is necessary to develop additional design criteria for primes of other forms.

In general, our experiments indicate that by choosing the encodings of the zero element such that the *don't care* terms (encodings of zero) are distributed evenly among the non-zero terms of $GF(p)$, the modulo multiplication function is minimized. We developed an easy method to accomplish this task. In particular, we built multiplication tables with $2^n \times 2^n$ entries (only $p - 1$ non-zero entries), and looked at how to divide it evenly into sections of non-zero values. Table 2 shows an example of the distribution of the zero-encodings that we found to be optimum for $p = 13$. This

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
1	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
2	•	•							•	•						
3	•	•							•	•						
4	•	•							•	•						
5	•	•							•	•						
6	•	•							•	•						
7	•	•							•	•						
8	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
9	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
10	•	•							•	•						
11	•	•							•	•						
12	•	•							•	•						
13	•	•							•	•						
14	•	•							•	•						
15	•	•							•	•						

Table 2: *don't care* distribution of best encoding for $p = 13$

allows us to give our second design criterion.

Design Criterion 2 *Choose the zero-encodings in such a way that they are distributed as evenly as possible throughout the $2^n \times 2^n$ entry multiplication table.*

It is obvious that there are many such *evenly distributed* encodings which might minimize the multiplier function. Notice that we don't claim that we £nd the encoding that minimizes the multiplication function over all possible encodings. Rather, we give an ef£cient search method to produce encodings which will potentially result in functions with reduced area complexities when compared with [5]. Our last heuristic is related to the way of choosing the encoding of the result of a multiplication by zero.

It is clear, that the encoding of a multiplication by zero has to belong to the set of zero encodings. However, which of the 2^n-p+1 possible encodings should one choose. This is particular interesting in the case in which the number of possible encodings is not a power of two. Criteria 3 and 4 provide us with guidelines to choose the encodings.

Design Criterion 3 *Let $D = 2^n-p+1$ be equal to 2^s for some s. Then, choose sets of 2^s zero encodings by £xing s of the n possible bits to be equal to 0 and setting the remaining ones to be don't cares (−). If setting s bits equal to 0 does not reduce the area complexity of the resulting function, then set them equal to 1. If that still does not work, try combinations of 0's and 1's.*

Design Criterion 4 *Assume $D = 2^n-p+1 \neq 2^s$. Then choose a set of $2^s < D$ zero encodings for the largest possible s and follow*

Criterion 3. Choose the remaining don't cares ($D - 2^s$) so as to satisfy Design Criterion 2. The encoding for any multiplication by zero result will correspond to the encoding of the set with 2^s elements.

As a £nal remark, we notice that in some cases setting the *don't cares* to zero, can further minimize the area complexity of the resulting function.

3.3. Complexity of Proposed $GF(p)$ Multipliers

The criteria of Section 3.2 was used to come up with encodings that would reduce the complexity of $GF(p)$ multipliers for $p < 2^5$. Once the possible encodings are chosen[2], we use ESPRESSO and SIS as explained in Section 3.1. After Boolean minimization, we use SIS once more for technology mapping with the *stdcell2_2.genlib* CMOS cell library. Then, we choose the encoding that realizes the multiplier with the least area. The area reported by SIS is a relative £gure obtained from the layout of the standard cells. We divide this value by the relative size of a pulldown or pullup (a pulldown/pullup is implemented with *one* transistor) according to the *stdcell2_2.genlib* library to obtain transistor counts. The number of transistors required to implement the logic functions for $2 < p < 2^5$, together with the encoding that gave the best results are summarized in Table 3. We have also implemented

p	Area [5]	Area new	%	ZERO element Encoding	Zero Encoding of Result
3	14	9	35.7	$(0)_2, (1)_2$	$'0-'$
5	65	42	35.4	$(0)_2, (1)_2, (2)_2, (3)_2$	$'0--'$
7	114	111	2.6	$(0)_2, (7)_2$	$'000'$
11	445	375	15.7	$(8)_2, (9)_2(10)_2,$ $(11)_2, (14)_2(15)_2,$	$'1-1-'$
13	566	520	8.1	$(0)_2, (1)_2(8)_2, (9)_2$	$'-00-'$
17	1048	907	13.5	$(0)_2, (1)_2(2)_2, (3)_2, (4)_2$ $(5)_2, (6)_2(7)_2, (8)_2, (9)_2$ $(10)_2, (11)_2(12)_2, (13)_2$ $(14)_2, (15)_2$	$'0----'$
19	1450	1343	7.4	$(0)_2, (2)_2(4)_2, (6)_2, (8)_2$ $(10)_2, (12)_2, (14)_2(16)_2,$ $(18)_2, (20)_2, (22)_2(24)_2,$ $(26)_2$	$'0---0'$
23	1984	1830	7.3	$(0)_2, (3)_2, (7)_2, (11)_2,$ $(15)_2, (19)_2, (23)_2,$ $(27)_2, (30)_2, (31)_2$	$'---11'$
29	3272	3136	4.2	$(4)_2, (12)_2, (20)_2, (28)_2$	$'00100'$
31	3806	3689	3.1	$(0)_2, (31)_2$	$'00000'$

Table 3: Area (transistors) complexity of proposed $GF(p)$ multipliers with given encodings.

and mapped the modulo multipliers proposed in [5] using the *stdcell2_2.genlib* library of SIS. We observe that in all cases our new encoding reduces the area complexity of the multiplier when compared to [5]. Although, we don't do so explicitly here, when compared to other multipliers in the literature, our new design outperforms all previous ones except for the one presented in [7].

3.4. Other Considerations

We notice that we have also implemented and mapped the circuits required to convert to/from our modi£ed representation. The area requirements indicate that our design will only be area ef£cient in cases where one needs many modulo p multipliers for the same p and at the same time one only needs a very few conversion circuits. One such application is $GF(p^m)$ multipliers for cryptographic applications where one requires 160–1024 bit long operands. For example, the cryptosystem described in [6] works over the £eld $GF(3^{163})$. In such a £eld, one would require 163 $GF(3)$ multipliers to implement a $GF(3^{163})$ multiplier. If one uses a single conversion circuit (since we process serially the coef£cients of the multiplicand), the conversion circuit would contribute less than one transistor per $GF(3)$ multiplier and thus, we can ignore it. Another important consideration is that in implementing a $GF(p^m)$ multiplier, we only need to change to and from our modi£ed representation at the beginning and at the end of the £eld multiplication (i.e. at system input/output times). Addition and subtraction which are also required in a $GF(p^m)$ multiplier can also be done in our modi£ed representation at no extra area penalty when compared to a normal $GF(p)$ adder/subtracter.

[2]Following the design criteria of Section 3.2, we never had to try more than 25 encodings before £nding one that was better than the encoding of [5].

4. REFERENCES

[1] D. V. Bailey and C. Paar. Ef£cient arithmetic in £nite £eld extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[2] M.A. Bayoumi, G. A. Jullien, and W.C. Miller. VLSI implementation of residue adders. *IEEE Transactions on Circuit and Systems*, CAS-34:284–288, March 1987.

[3] D. Boneh and M. Franklin. Identity-Based Encryption from the Weil Pairing. In J. Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, volume LNCS 2139, pages 213–229. Springer-Verlag, 2001.

[4] E. D. Di Claudio, F. Piazza, and G. Orlandi. Fast Combinatorial RNS Processors for DSP Applications. *IEEE Transactions on Computers*, 44(5):624–633, May 1995.

[5] A. A. Hiasat. Semi-Custom VLSI Design for RNS Multipliers Using Combinatorial Logic Approach. In *Third IEEE International Conference on Electronics, Circuits, and Systems — ICECS '96*, pages 935–938, October 13–16, 1996.

[6] N. Koblitz. An elliptic curve implementation of the £nite £eld digital signature algorithm. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO 98*, volume LNCS 1462, pages 327–337. Springer-Verlag, 1998.

[7] V. Paliouras, K. Karagianni, and T. Stouraitis. A Low-Complexity Combinatorial RNS Multiplier. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 48(7):675–683, July 2001.

[8] D. Radhakrishnan and Y. Yuan. Novel Approaches to the Design of VLSI RNS Multipliers. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(1):52–57, January 1992.

[9] N. Smart. Elliptic Curve Cryptosystems over Small Fields of Odd Characteristic. *Journal of Cryptology*, 12(2):141–151, Spring 1999.

[10] D. J. Soudris, V. Paliouras, T. Stouraitis, and C. E. Goutis. A VLSI Design Methodology for RNS Full Adder-Based Inner Product Architectures. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 44(4):315–318, April 1997.