# Novel Multiplier Architectures for $GF(p)$ and $GF(2^n)$

E. Savaş[1], A. F. Tenca[2], M. E. Çiftçibasi[3], and Ç. K. Koç[2]

[1] Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul, Turkey TR-34956

[2] Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331

[3] Aselsan Inc.
Ankara, Turkey

**Abstract**

This paper proposes two new hardware architectures for performing multiplication in $GF(p)$ and $GF(2^n)$, which is the most time consuming operation in many cryptographic applications. The architectures provide very fast and efficient execution of multiplication in both $GF(p)$ and $GF(2^n)$, and can be mainly used in elliptic curve cryptography. Both architectures are scalable and therefore can handle operands of any size. They can be configured to the available area and/or desired performance. The algorithm implemented in the architectures is Montgomery multiplication algorithm which proved to be very efficient in both fields. The first architecture utilizes a precomputation technique that reduces the critical path delay at the expense of using extra logic which has a limited negative impact on the silicon area for operand precisions of cryptographic interest. The second architecture computes multiplication faster in $GF(2^n)$ than $GF(p)$, which conforms with premise of $GF(2^n)$ for hardware realizations. Both architectures provide new alternatives that offer faster computation of multiplication and useful features.

**Index terms:** Montgomery multiplication, unified and scalable architecture, dual-radix architrave, public-key cryptography.

# 1 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, 5].

The Montgomery multiplication algorithm is originally proposed as an efficient method for doing modular multiplication with an odd modulus [6]. When the modulus is a prime number, the Montgomery multiplication algorithm becomes a very efficient tool to perform multiplication in prime fields $GF(p)$. 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. Furthermore, since the Montgomery algorithm applies the reduction operation (when it is necessary) to the partial result starting from the least significant digit, which is opposite to the order of the classical modular multiplication algorithm, it eliminates the need to await the completion of carry propagation before starting working with the next digit of the multiplier. 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 [7, 8, 9] and pipeline organizations of multiple atomic processing units in order to exploit the inherent parallelism in the algorithm [10]. Systolic array-based implementations such as [9] achieve very high throughput rates when there are many successive multiplication operations.

In [11], it is also shown that Montgomery multiplication might be very efficient in $GF(2^n)$, when polynomial basis is used and the irreducible field polynomial is chosen arbitrarily. 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 [10, 12, 13].

Various hardware implementations of the Montgomery multiplication algorithm for limited precision operands were proposed in [14, 15, 16]. Implementations utilizing high-radix modular multipliers have also been proposed in [15, 17, 18, 19]. Aspects of using high-radix representation have been discussed in [20]. There are certain trade-offs in speed and circuit complexity for high-radix architectures. Even though very high-radix designs have certain complications in hardware, moderate radix values offer faster alternatives to simple radix-2 multiplier designs. For instance, high-radix multipliers proposed in [21, 22] for binary extension fields $GF(2^n)$ are attractive for both low-power and high-performance applications.

The original unified multiplier in [10] 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 $GF(p)$ at the expense of using some amount of extra gates without significantly increasing the signal propagation time.

In this paper, following the design principles introduced in [10], we present novel unified and scalable multiplier architectures utilizing two original design ideas. The first one is to employ an online precomputation method that eliminates a computation that is performed several times in the previous architectures, hence reducing complexity and the critical path delay in the architecture, at the expense of additional registers. And the second one is a multiplier module working with radix-$2^{k+1}$ for $GF(2^n)$ and radix-$2^k$ for $GF(p)$. 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.

The rest of the paper is organized as follows. Section 2 gives the definition of the three key concepts: scalable and unified architecture, and dual-radix design. It also contains very short explanation about the fundamentals of prime and binary extension fields and arithmetic operations in these fields. It is followed by an introduction to Montgomery multiplication algorithm and its usage in finite field arithmetic. Section 3 introduces the first member of the dual-radix multiplier family, radix-(2,4) multiplier. Besides the design details and the components of the radix-(2,4) multiplier, comparative complexity analysis and performance evaluation are given in this section. The implementation results are summarized in Section 4. In Section 5, new architecture is compared against previous architectures and the conclusion is given in Section 6.

# 2 Preliminaries

## 2.1 Definitions

**Scalable Architecture :** An arithmetic unit is called *scalable* if it can be reused or replicated in order to generate long-precision results independently of the data path precision for which the unit was originally designed. One can profitably refer to [23] for a thorough discussion about the advantage of scalable hardware.

**Unified Architecture :** An architecture is said to be *unified* when it is able to work with operands in both prime and binary extension fields, $GF(p)$ and $GF(2^n)$, using the same datapath.

In [10], it has been shown that a unified multiplier is feasible with only minor modifications to the scalable multiplier for $GF(p)$ in [23][1]. In the unified multiplier [10], it has been demonstrated that the propagation time is unaffected while the increase in chip area is insignificant.

**Dual-Radix Architecture :** A unified multiplier is said to be *dual-radix* if it operates with a larger radix value for $GF(2^n)$ than the radix used for $GF(p)$.

A dual-radix unified multiplier must be designed in such a way that the cost of this extra functionality on chip area and the propagation time of the critical path must be of acceptable levels.

[1] Note that a $GF(2^n)$-only multiplier can always be designed to outperform a unified design.

## 2.2 Fundamentals of prime and binary extension fields

An elliptic curve can be constructed over different mathematical entities such as a ring or field. However, only finite fields are used in cryptographic applications because of their suitability to implementation in digital systems. Especially the prime field $GF(p)$ and binary extension field $GF(2^n)$ become favorable since various standard bodies such as National Institute of Standards and Technology (NIST) and American National Standards Institute (ANSI) specifically recommends several elliptic curves over these finite fields.

The elements of the prime finite field $GF(p)$ are the integers $\{0, 1, 2, ..., p-1\}$ where $p$ is an odd prime. The addition and multiplication operations in $GF(p)$ are modular operations performed in two steps: (1) regular integer addition or multiplication and (2) reduction by the prime modulus $p$ if the result of the first step is greater than or equal to the modulus.

The elements of the binary extension field $GF(2^n)$ can be represented as binary polynomials of degree less than $n$ if polynomial basis representation is used. Analogous to the odd prime used in $GF(p)$, a binary irreducible polynomial of degree $n$ is used to construct $GF(2^n)$. The addition in $GF(2^n)$ is simply done by modulo-2 addition of corresponding coefficients of two polynomials. Since it is basically a polynomial addition there is no carry propagation and the degree of the resulting polynomial cannot exceed $n$. On the other hand, multiplication in $GF(2^n)$ is more complicated and sometimes it is beneficial to use other type of representation techniques than standard polynomial basis such as Gaussian normal basis [24]. In this paper, we always use polynomial basis for $GF(2^n)$ because of its suitability to the unified architecture.

Polynomial basis representation of $GF(2^n)$ is determined by an irreducible binary polynomial p(x) of degree n. Given $p(x)$, all the binary polynomials of degree less than $n$, which has the form $A(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$, are elements of $GF(2^n)$. Multiplication in $GF(2^n)$, similar to multiplication in $GF(p)$, is performed in two steps: (1) polynomial multiplication followed by (2) a polynomial division of the result from Step 1 by the irreducible polynomial $p(x)$ [2].

## 2.3 Montgomery Multiplication Algorithm

In [6], 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. And opposite to the subtraction of modulus in the regular modular multiplication which can be performed after all the digits of the multiplicand are processed, the addition operation can start immediately after the least significant digit of the multiplicand is processed. Especially the second feature accounts for the inherent concurrency in the algorithm. Refer to [6, 25, 26] For detailed explanation of the algorithm.

[2] In general, these two steps are interleaved in implementation.

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 $c$. These transformations are applied as in the following example:

$$\bar{a} = MonMult(a, R^2) = a \cdot R^2 \cdot R^{-1} \pmod{p} = a \cdot R \pmod{p},$$

$$\bar{b} = MonMult(b, R^2) = b \cdot R^2 \cdot R^{-1} \pmod{p} = b \cdot R \pmod{p}$$

$$c = MonMult(\bar{c}, 1) = c \cdot R \cdot R^{-1} \pmod{p} = c \pmod{p}.$$

Provided that $R^2 \pmod{p}$ is precomputed and saved, we need only a single *MonMult* operation to carry out each of these transformations. However, because of these transformation operations, performing a single modular multiplication using *MonMult* might not be advantageous even though there is an attempt to make it efficient for a few modular multiplications by eliminating the need for these transformations [27]. Its advantage, on the other hand, becomes obvious in applications requiring multiplication-intensive calculations such as modular exponentiation and elliptic curve point operations.

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: | $c \in [1, p-1]$ |
| Step 1: | $c := 0$ |
| Step 2: | for $i = 0$ to $m-1$ |
| Step 3: | $q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$ |
| Step 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 $j$th bit in the $i$th 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: | $c(x)$ |
| Step 1: | $c(x) := 0$ |
| Step 2: | for $i = 0$ to $m - 1$ |
| Step 3: | $q(x) := (c_0(x) + a_i(x) \cdot b_0(x)) \cdot p'_0(x) \pmod{x^k}$ |
| Step 4: | $c(x) := (c(x) + a_i(x) \cdot c(x) + q(x) \cdot p(x))/x^k$ |

where $p'_0(x) = p_0^{-1}(x) \pmod{x^k}$. As one easily observes the two algorithms are almost identical except that the addition operation in $GF(p)$ becomes a bitwise modulo-2 addition in $GF(2^n)$. Although the operands are integers in the former algorithm and binary polynomials in the latter, the representations of both are identical in digital systems. 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 [28, 29], hence we intentionally exclude it from the algorithm definitions.

One can also observe that the computations performed in Step 3 are of different nature in two algorithms and depending on the magnitude of the radix used, the part of the circuit in charge of implementing them might become very complicated. However, we demonstrate in the subsequent sections that these computations can be performed in a unified circuitry for small radices.

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.

## 2.4 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 [10]. Note that Step 4 of the

Algorithm A computes

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

where division by $2^k$ is simply a right shift by $k$ bits and $q$ is calculated in the previous step. Depending on the radix value chosen for the multiplier, the $k$-bit digit $q$ can be determined by the least significant digits (LSD) of $b$, $p$ and $c$, and the current digit of $a$. Similarly, the multiple of $b$ that participates in the addition is determined solely by $a_i$. As a result, 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.

# 3 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. It is called radix-(2,4), and it is, in fact, the first member of the dual-radix multiplier family whose other members may be radix-(4,8) and radix(8,16). We only included radix-(2,4) multiplier for the sake of simplicity and for another member of the family, radix-(4,8) multiplier, the architecture and selection logic are given in Appendix B.

The radix-(2,4) (or *dual-radix*) multiplier utilizes the precomputation technique presented in the previous section.

## 3.1 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:

Step 3: $q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$

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

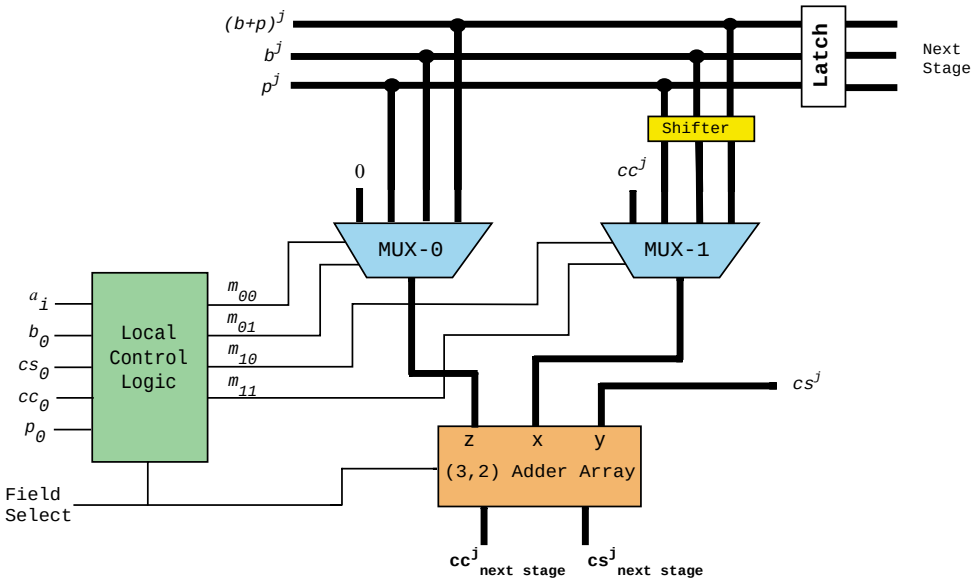


Figure 1: Processing unit of dual-radix architecture with radix-2 for $GF(p)$ and radix-4 for $GF(2^n)$

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: $\{0, b, p, b+p, x \cdot b, x \cdot p, x \cdot (b+p)\}$ (Recall that $a_{i,j}$ is the $j$th least significant bit of $i$th digit of $a$). Multiplication by $x$ results in shifting one bit to the left, hence it is identical to multiplication by 2. Throughout the paper division by $x^k$ and $2^k$ are identical operations and the latter is used to denote the right shift operation by $k$ bits.

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, $m_{00}$, $m_{01}$, $m_{10}$, and $m_{11}$, to determine which multiples of $b$ and $p$ will be in the calculations. For example, $m_{00}m_{01}m_{10}m_{11} = (1011)$ indicates that Step 4 will be $c := (c + 3b + 2p)/2^k$. We will give the implementation details of the selection logic in subsequent sections. $cc_0$ 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

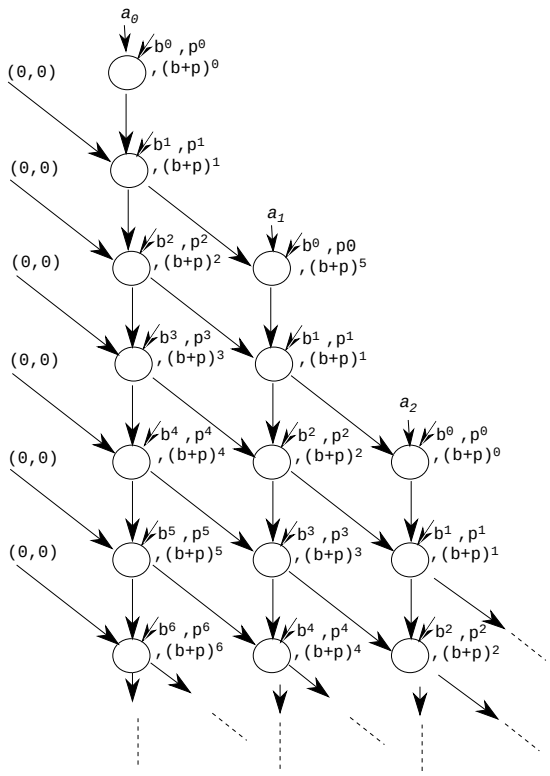


Figure 2: Execution graph of Montgomery multiplication algorithm [23]

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. When the last PU in the pipeline starts generating the partial results, the control circuitry checks if the first PU is available. If the first PU is still working on an earlier computation, the results from the last PU should be stored in a buffer until the first PU becomes available again. One can refer to [10] for more information about the length of the buffer to store the partial results when there is no available PU in the pipeline. In Figure 2 the execution graph of the Montgomery multiplication algorithm is illustrated. An example of pipeline organization with two PUs is shown in Figure 3.

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. The partial result $c$ is kept in redundant form during the computations and it must be converted to non-redundant form when the multiplication is completed. In addition, one must note that the length of the register for partial result in Figure 3 is twice wider than the other registers.

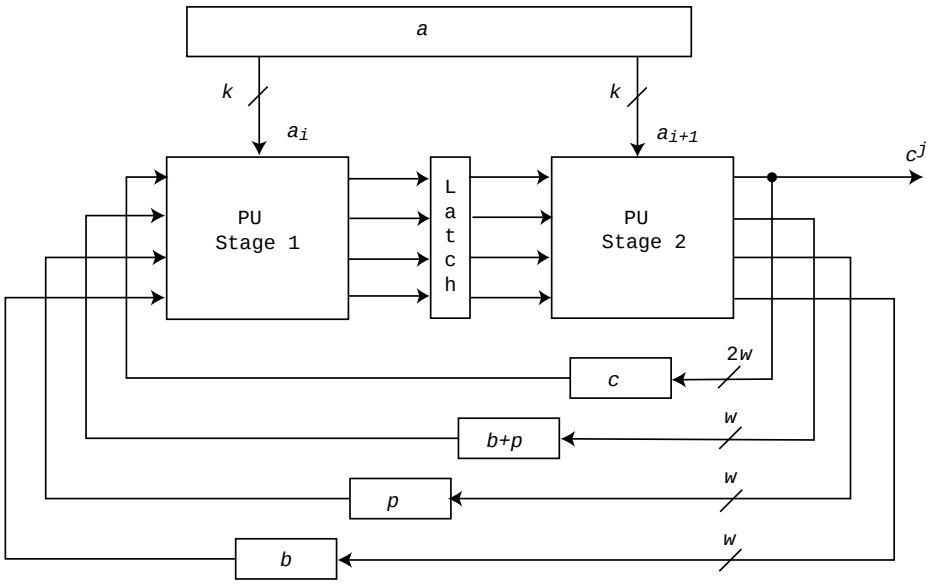


Figure 3: Pipeline organization with two processing units

Given that carry-save notation is used for the partial result and that each iteration is executed on word-by-word basis, the Algorithm A can be expressed as follows:

**Algorithm A (modified)**

| | |
|---|---|
| Input: | $a, b \in [1, p-1]$, $p$, and $m$ |
| Output: | $c \in [1, p-1]$ |
| Step 1: | $c := 0$ |
| Step 2: | for $i = 0$ to $m-1$ |
| Step 3: | $q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$ |
| Step 4: | for $j = 0$ to $e-1$ |
| Step 5: | $c^{(j)} := (cc^{(j)} + cs^{(j)} + a_i \cdot b^{(j)} + q \cdot p^{(j)})/2^k$ |

While a redundant format enables us to employ Carry-Save adders, which are typically less costly in terms of area and delay than carry propagate adders, it 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. Implementation details of such an adder for performing addition both in $GF(p)$ and $GF(2^n)$ can be found in [30], where it has been shown that the critical path delay of the adder is lower than that of a PU in [10] and its area does not exceed 39% of that of a PU for the same word length. Considering that several PUs in a multiplier are used for cryptographic applications (e.g. five PUs at least

for 160-bit multiplications) the overhead of the adder in the area becomes insignificant when it is compared to the whole multiplier.

The multiplier, after certain number of clocks elapsed, generates one word of the result in each clock cycle starting from the least significant word. The adder, which operates on the words of the result, immediately starts adding the available word of the result in the next clock cycle after the word is produced at the output of the multiplier. When the same clock speed and word length are used for both multiplier and adder, the conversion and multiplication operations are overlapped, thus the total cost of this addition operation in terms of time is only an extra cycle at the end. One can profitably refer to [30, pages 38–41] for a better understanding of the overlapping technique and characteristics of the adder.

In case the word length of the adder is less than that of multiplier due to a possible mismatch in the critical path delays of the multiplier and adder, a shorter word adder may have to be used. In this case, the cost of the conversion operation will be more than one extra cycle and it can be formulated depending on the difference between the word sizes of the adder and multiplier. In the worst case when there is no overlapping, the total cost of conversion is $t_{conv} = \lceil n/w_{adder} \rceil$ where $n$ is the bitsize of the modulus and $w_{adder}$ is the length of the adder. Comparing the total number of clock cycles to compute one multiplication, which is about $2n$, the cost of the conversion becomes insignificant for relatively long adders (e.g. $w \geq 8$).

The existence of an adder is also useful for performing the precomputation of $b + p$, which is used during multiplication. As in the addition operation that transforms the result into non-redundant form, the precomputation costs only an extra clock cycle before the multiplication operation starts. The extra cost for the precomputation in terms of area comes from the extra register to store $b + p$ and one extra inter-stage $w$-bit register for the partial result words generated by PUs. However, since the precomputation technique eliminates the second layer of carry-save adders in [10], this increase is partially compensated. The actual effect of the precomputation technique on area and critical path delay is discussed in Section 4.

### 3.2 (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 4. When used in $GF(p)$-mode, the DFA simply becomes a Carry-Save adder.

In Figure 5, typical implementation of a DFA cell is illustrated. 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. When $FSEL = 1$, it performs bitwise addition with carry which enables the multiplier to do arithmetic in $GF(p)$. When $FSEL = 0$, on the other hand, the carry output is forced to 0 regardless of the input values. An important aspect of designing a DFA cell is to avoid increasing the critical path of the circuit with respect to full-adder, which can have an adverse effect on the clock speed and it is against our design premise. The area and signal propagation aspects of a DFA cell are almost identical to those of a typical full-adder which would

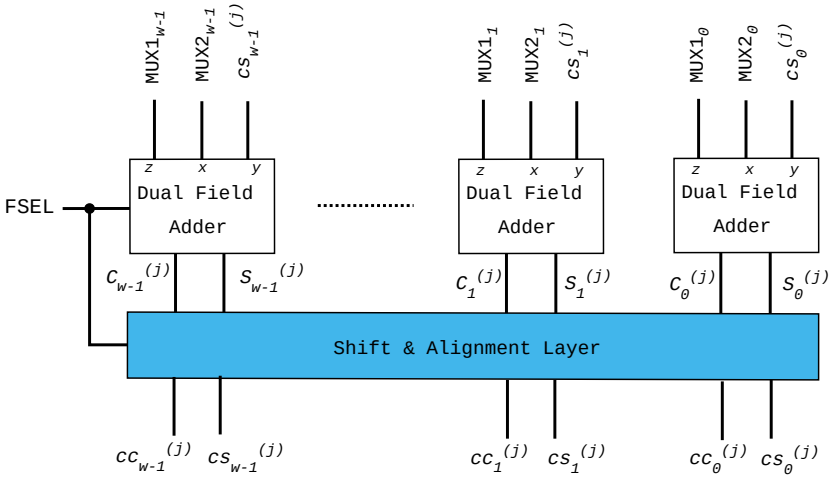


Figure 4: Dual Field Adder Array for radix-(2,4) Unified Multiplier

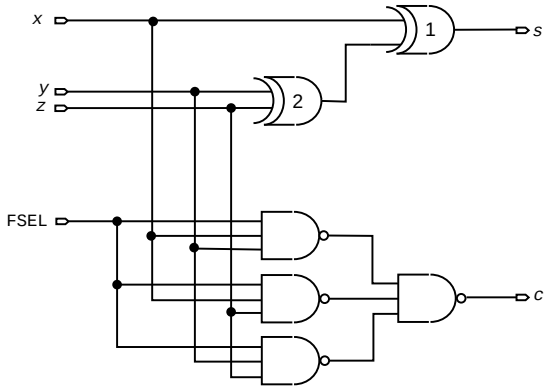


Figure 5: Dual-Field Adder Cell

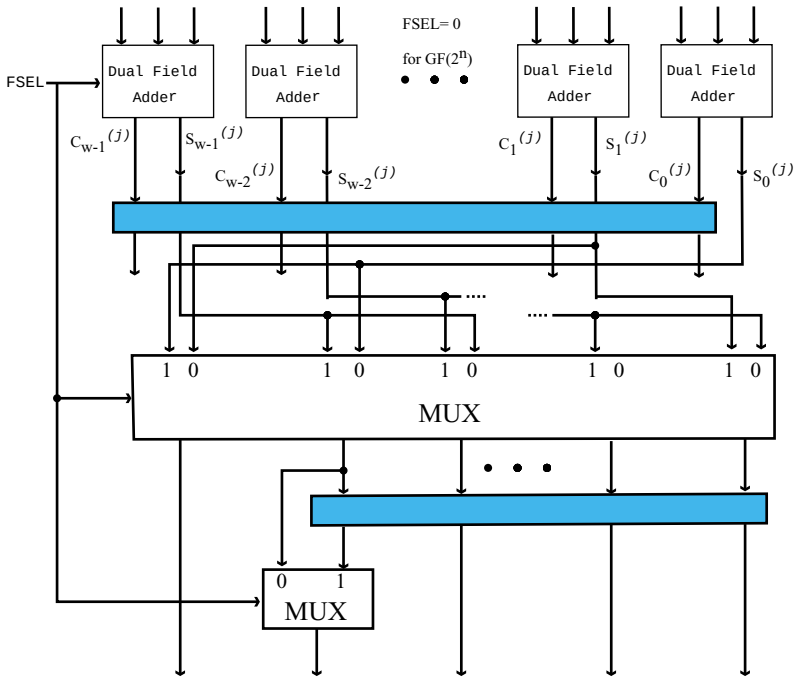


Figure 6: Shift and Alignment Layer

be used in a $GF(p)$-only multiplier. Therefore, this additional functionality is obtained almost without any cost.

The shift and alignment layer performs different operations depending on the field. When in $GF(p)$, a single-bit shift to the right must be performed on all the words coming from the dual-field adder. Let us call these words $S^{(j)} = (S^{(j)}_{w-1}, ..., S^{(j)}_1, S^{(j)}_0)$ and $C^{(j)} = (C^{(j)}_{w-1}, ..., C^{(j)}_1, C^{(j)}_0)$ (sum and carry vectors that represent word $j$). The adder output vector $C^{(j)}$ is already shifted one position to the right with respect to the sum vector $S^{(j)}$. Therefore, the word $cs^{(j)} = C^{(j)}$ could in principle be sent to the next module in the same clock cycle, however, $cs^{(j)} = (S^{(j+1)}_0, S^{(j)}_{w-1}, ..., S^{(j)}_1)$ and for this reason all the bits of $C^{(j)}$ and the bits $w-1$ to 1 of $S^{(j)}$ must be delayed to be combined with the least significant bit of word $S^{(j+1)}$. The situation is quite similar for $GF(2^n)$, but in this case: (i) the $C$ output of the adder is always zero and (ii) a 2-bit right shift is required. Thus, in this case, $cc^{(j)}$ is not important, and $cs^j = (S^{(j+1)}_1, S^{(j+1)}_0, S^{(j)}_{w-1}, ..., S^{(j)}_2)$. This conditional mapping of the $cs^j$ output bits, which depends on the field being used, is implemented by the two multiplexers and registers shown in Figure 6.

## 3.3 Selection Circuitry

As stated previously, the selection logic for radix-(2,4) multiplier, which is shown in Figure 7, 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

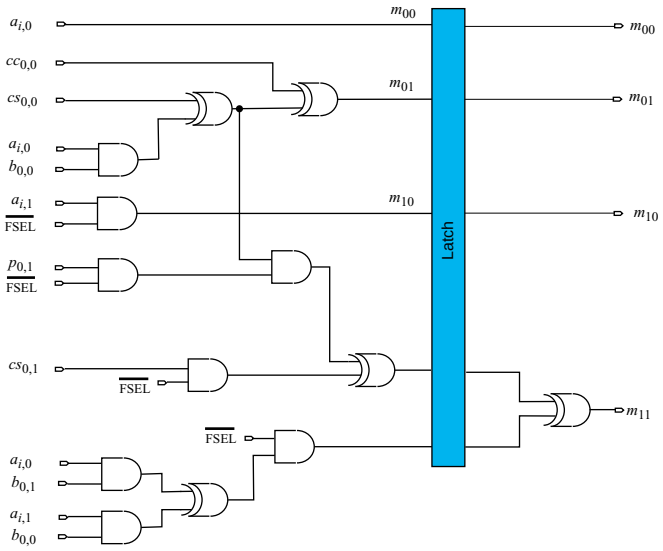


Figure 7: Selection logic for radix-(2,4) multiplier

calculates $c := c + a_i \cdot b + q \cdot p$. The selection logic constitutes a major section of the local control. For more information on the derivation of the formulae that the selection logic implements, see Appendix A. In addition, for quick reference, the selection inputs of the multiplexers are tabulated in Table 1.

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 $j$th bit of $q$, and $cs_{i,j}$ and $cc_{i,j}$ are the sum and carry bits of the partial result, respectively.

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.

As can easily be seen in Figures 1, 5, and 7, there are 3 XOR and 2 AND gates in the critical path of the selection logic while the critical path of a PU is determined by the $(3, 2)$ adder delay ( dominated by

| | $GF(p)$-mode | $GF(2^n)$-mode |
|---|---|---|
| $m_{00}$ | $a_{i,0}$ | $a_{i,0}$ |
| $m_{01}$ | $(cs_{0,0} \oplus cc_{0,0} \oplus a_{i,0} \cdot b_{0,0})$ | $(cs_{0,0} \oplus a_{i,0} \cdot b_{0,0})$ |
| $m_{10}$ | 0 | $a_{i,1}$ |
| $m_{11}$ | 0 | $(cs_{0,1} \oplus a_{i,0} \cdot b_{0,1} \oplus a_{i,1} \cdot b_{0,0}) \oplus (cs_{0,0} \oplus a_{i,0} \cdot b_{0,0})$ |

Table 1: Selection Inputs to MUX-0 and MUX-1

two serially connected XOR gates) and some delay due to the multiplexers (multiplexer delay is usually less than a XOR gate). The computation of $m_{00}, m_{01}, m_{10}$ and $m_{11}$ needs to be performed at the first clock cycle when a PU starts processing. However, the computation regarding the selection logic can be done at a separate clock cycle at the beginning, before a PU starts its actual computation. This does not increase the number of clock cycles between the two consecutive PUs in the pipeline, which is 2 in the original design, since the inputs to the selection logic of a PU become ready at the end of the first clock cycle of the previous PU. Therefore, this extra selection cycle at the very beginning increases the total number of the clock cycles to perform one multiplication by only one clock cycle independent of the precision of the operands.

The selection logic dominates the critical path since its 3 XOR and 2 AND gate delay is greater than that of the core part of a PU. Therefore, the selection logic is divided into two parts as seen in Figure 7. As the first part of the computation is done in the selection cycle (the very first cycle) the second part can be performed in the following cycle. The second part introduces one XOR gate delay to the circuitry, whose effect is compensated by connecting the output of the MUX-1 to the input $x$ of the dual field adder. This approach avoids serially connecting the XOR gate in the second part of the selection logic to the second XOR gate of the dual field adder. This implementation detail prevents the selection logic from dominating the critical path of the design. The gate level realization of the selection logic and its division into two parts, to some extent, depend on the technology being used and with careful implementation the delay of first part can almost always be made less than the core part of a PU.

In Figure 2 observe that a processing unit, $\text{PU}_i$ generates all the bits of a partial result word except the leftmost $k$ bits in each clock cycle and wait for another cycle for the rest of the leftmost $k$ bits before passing the partial result word to $\text{PU}_{i+1}$ in the next stage. This explains the two clock cycle delay between the starting times of two consecutive pipeline stages. In this design, although $\text{PU}_{i+1}$ waits two clock cycles to start calculation, its selection logic starts processing some of the available rightmost bits (exact number depends on the radix) arrived from $\text{PU}_i$. A similar technique, called *retiming*, is explained in detail in [31, 19].

### 3.4 Complexity Analysis and Performance

There are three criteria to assess the performance of our multiplier design:

- **Maximum clock frequency** The signal propagation time of the critical path of a single PU, to a large extent, determines the maximum clock frequency that can be applied to the multiplier. It is a desirable feature to be able to operate the multiplier at higher frequencies when it is possible. As explained in 3.3, the delay of the selection logic is designed to be smaller than that of the main computation block of a PU, which mainly consists of a multiplexer and (3,2) adder array. As opposed to the original unified multiplier design in [10] in which a PU has two layers of adder arrays, our new design employs PUs with only one layer of adder arrays. Considering that the two XOR gates in one layer of adder arrays dominates the critical path of a PU, the new multiplier design theoretically allows twice the clock frequency of the original design. However, one should take into account possible extra wiring delays that might originate, for example, from the shifting of three operands $p$, $b$ and $b+p$. In addition, it is obvious that Shift-and-Alignment layer also have a negative impact on the critical path delay.

  Nevertheless, it would be safe to say that eliminating one layer of dual-field adders from each PU improve the maximum clock frequency.

- **Number of clock cycles** The total computation time of a multiplication operation is determined by the number of clock cycles, which in turn depends on the underlying algorithm the unit implements. In the new design, the number of clock cycles is determined by (1) the operand precision $n$, (2) the operating mode ($GF(p)$ or $GF(2^n)$), (3) the word size $w$, and (4) the number of pipeline stages $t$ (the number of PUs in the pipeline).

  Each column in the graph shown in Figure 2 represents operations that can be performed by separate PUs in the pipeline. As explained in 3.1, each PU completes its share of work for a digit of the multiplier $a$ after $e+1$ clock cycles, where $e$ is the number of the words in the modulus and becomes available for processing another digit of $a$. In case there is no available PU and there are more digits of $a$ to be processed, the data generated by the last PU waits in a buffer for the first PU to finish its job. Therefore, the $c$ buffers in Figure 3 are provided to store the partial results until the first PU becomes free. The length requirement of these extra buffers, which depends on the precision and the exact configuration of the pipeline organization in terms of number of stages and word size, is out of scope of this paper and extensively studied in [10]. If there are at least $\lceil (e+1)/2 \rceil$ PUs in the pipeline organization the extra buffers are not needed. Considering these extra cycles the data need to wait, the total computation time in terms of number of clock cycles is given as:

$$T = \begin{cases} 2t \cdot \lceil \frac{n}{kt} \rceil + e + 1 & \text{if } (e+1) < 2t \\ \lceil \frac{n}{kt} \rceil \cdot (e+1) + 2t & \text{otherwise} \end{cases} \quad (1)$$

where $t$ is the number of PUs, $e$ is the number of words, $k$ is the radix value (1 for $GF(p)$ and 2 for $GF(2^n)$), and $n$ is the modulus precision. Note that one extra cycle at the beginning for calculating the first word of $b+p$ and selection values, and one extra clock cycle at the end for transforming the redundant result to non-redundant form are also included in the formula.

In the new design, the total computation time in $GF(2^n)$-mode is almost half of the original design since radix-4 (twice the radix in the original design) is used while the total computation time remains identical to [10] in $GF(p)$-mode.

- **Area** To measure the size of VLSI implementation of a module we use the number of gates which would take the same amount of circuit area as the module. The choice of gate for this equivalency depends on the technology used. While the gate count of a PU cell is determined largely by the word size, the area of whole pipeline organization depends also on the number of pipeline stages, local control logic, inter-stage latches and the registers to hold the operands including those necessary when pipeline stalls occurs.

  In the new design, the PU cell area is reduced since the second layer of adder array is no longer necessary. On the other hand, local control logic becomes more complicated due to the selection logic and precomputed value $b+p$ that must be hold in inter-stage latches. Since there are already 4 inter-stage latches of length $w$ in the original design (considering the partial result is treated in two separate parts) we increase the register size in PU by 20% by adding one extra latch of length $w$ for $b+p$. Also, a more complicated Shift-and-Alignment layer is expected to increase the area requirement of a single PU.

  The exact overall effect of reduction in the PU cell size and increase in the size of local control logic, and the registers and latches depends on the design choices such as word size, number of stages etc.

# 4 Implementation results

We implemented processing units of three different multiplier architectures: **(A1)** the original unified multiplier in [10], **(A2)** single-radix multiplier utilizing precomputation techniques, and **(A3)** radix-(2,4) multiplier. We used VHDL to implement three architectures and synthesized the resulting code using Mentor Graphics tools for an ASIC technology of $0.5\mu m$ AMI CMOS (ADK library [32]). During the synthesis, we adopted the *Preserve Hierarchy* [1] option.

Figure 8 demonstrates the area and time delay of three different PU designs, using different word sizes. Area consumption is always given in terms of 2-input **NAND** gates. Note that the actual area requirements

[1] preserve hierarchy option prevents the CAD tool from flattening design, having the effect that synthesis results (time and area) do not change depending on the number of the PUs.

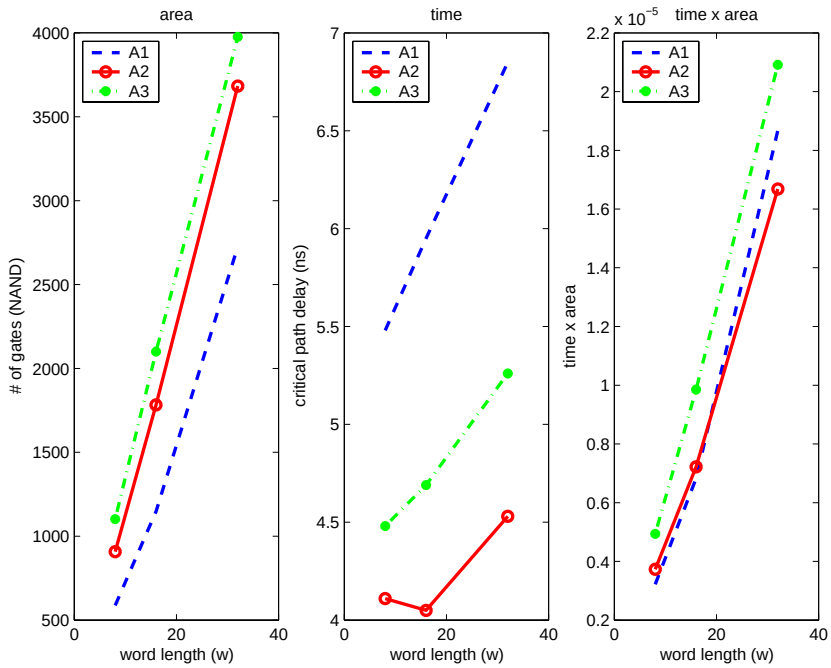


Figure 8: Implementation results: Critical path delay and area

depend not only on the word size but also the number of PUs used in the pipeline organization. Detailed analysis and necessary formulation are provided for the original unified architecture in [10], which identically apply to the two new designs introduced in this paper. 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.

Besides these three unified multiplier architectures, there is also $GF(p)$-only multiplier architecture introduced in [23]. An extensive comparison of $GF(p)$-only multiplier against a simple unified architecture is done in [10] and it has been found that the critical path delay of the unified architecture is almost the same as that of $GF(p)$-only multiplier, while the increase in area is about 2.8%. Because of this insignificant increase in area as well as the fact that the comparison has already been done, we chose not to include the implementation results of $GF(p)$-only multiplier here.

As can easily be observed from Table 2, where the relative increases of chip areas of the two new designs with respect to basic unified architecture is shown, there is an increase in area of the new architectures. Common to both designs, 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. We compared logic and register area in three architectures, and confirmed that the register area is relatively higher in the two new architectures than the original unified design. 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. In the dual-radix case, 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 (see Figure 6) 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. The architecture **A2** uses only 35% more area than the basic unified design (**A1**) while this increase becomes, in the dual-field multiplier (**A3**), 45% when $w = 32$. It is also important to note that the additional functionality of operating with different radices in $GF(p)$ and $GF(2^n)$ of the dual-field multiplier comes with an increase of 8% to 21% in area.

| Word size (w) | A2/A1 | A3/A2 | A3/A1 |
|---|---|---|---|
| 8 | 55% | 21% | 88% |
| 16 | 55% | 18% | 83% |
| 32 | 35% | 8% | 45% |

Table 2: Relative increase in design area

Table 3 summarizes the relations in the maximum clock frequencies that can be applied to each architecture. The use of the precomputation technique in the architecture **A2** improves the critical path delay by 25% to 34%, while the improvement in the architecture **A3** is 18% to 23%. The relative deterioration of 8% to 14% in critical path delay in **A3** with respect to **A2** is due to a more complicated shift-and-alignment layer and use of two multiplexers (see Figures 1 and 6).

| Word size (w) | A2/A1 | A2/A3 | A3/A1 |
|---|---|---|---|
| 8 | 25% | 8% | 18% |
| 16 | 24% | 14% | 21% |
| 32 | 34% | 14% | 23% |

Table 3: Speedup in delay

The performance of all three 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. This can be easily observed in Equation 1. Therefore, the relative increase in the area of a PU may be misleading in evaluating the overall performance of the new architectures. All three architectures utilize many PUs organized in a pipeline. Although the architectures scale to any operand precision, a certain number of the PUs in the pipeline offers an optimized performance for only a specific range of operand precision. Due to pipeline stalls explained in [10], the performance of the multiplier

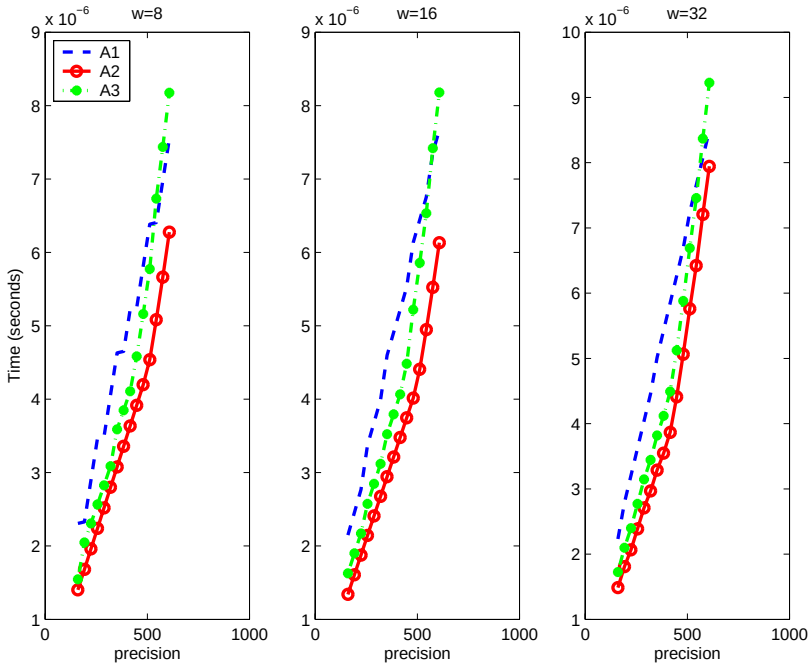


Figure 9: Multiplication Timings (in $\mu$s) for an area of 30.000 gates with $w = 8, 16$, and 32 in $GF(p)$-mode

| precision | w = 8 | | | w = 16 | | | w = 32 | | |
|---|---|---|---|---|---|---|---|---|---|
| (bits) | A1 | A2 | A3 | A1 | A2 | A3 | A1 | A2 | A3 |
| 160 | 2.3 | 1.4 | 1.5 | 2.1 | 1.3 | 1.6 | 2.2 | 1.5 | 1.7 |
| 192 | 2.4 | 1.7 | 2.0 | 2.5 | 1.6 | 1.9 | 2.8 | 1.8 | 2.0 |
| 228 | 2.9 | 2.0 | 2.3 | 2.8 | 1.9 | 2.2 | 3.2 | 2.1 | 2.4 |
| 256 | 3.5 | 2.2 | 2.6 | 3.4 | 2.1 | 2.6 | 3.6 | 2.4 | 2.8 |

Table 4: Multiplication Timings (in $\mu$s) for different precisions with 30,000 gates in $GF(p)$-mode

deteriorates for the operands beyond the specified operand precision. As the operand precision increases, it may become necessary to add more PUs to the pipeline, which will naturally leads to increase in chip area. The problem of the best area/time relationship has been extensively investigated in [33]. To provide more insight in the overall effect of the new architectures on the area and time, we investigated the time to compute multiplication for a precision range of cryptographic interest given a limited area. Figure 9 demonstrates the results for multiplier configuration in $GF(p)$-mode with approximately $30,000$ gates (which has been considered, in previous works on the architecture, as typical for an optimized performance in cryptographic applications). We basically designed the multipliers for each architectures by putting as many PUs as possible.

In this configuration, the new architectures, **A2** and **A3**, offer a significant speedup in time performance over the original architecture **A1** for the range of $[160, \sim 500]$. Note also that for this range of precision, **A2** performs better than **A3**. Beyond the precision of 500 bits, higher area requirements of new architectures will have a negative impact on the performance. In order to demonstrate the advantage of two new architectures, we listed the times needed to compute multiplication of operands for different precisions in Table 4. For the same area the new architecture, **A2** offers a speedup in time over **A1** by 26% to 39% for these different precisions in $GF(p)$-mode, while this speedup in case of **A3** is by 13% to 35%[3]. Note that the maximum speedup in both architectures, **A2** and **A3**, 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.

While the architecture **A3** performs slightly worse than **A2** in $GF(p)$-mode (between 7% to 19%), it offers a significant speedup over both **A2** and **A1** in $GF(2^n)$-mode. It outperforms **A1** by 56% to 67%, while the speedup over **A2** is 38% to 46% in this mode.

| precision | w = 8 | | | w = 16 | | | w = 32 | | |
|---|---|---|---|---|---|---|---|---|---|
| (bits) | A1 | A2 | A3 | A1 | A2 | A3 | A1 | A2 | A3 |
| 160 | 1.9 | 1.4 | 1.6 | 2.1 | 1.4 | 1.6 | 2.2 | 1.5 | 1.7 |
| 192 | 2.4 | 1.7 | 1.9 | 2.4 | 1.6 | 2.0 | 2.7 | 1.8 | 2.4 |
| 228 | 2.7 | 2.0 | 2.6 | 2.8 | 2.0 | 2.7 | 3.1 | 2.0 | 3.2 |
| 256 | 3.2 | 2.6 | 3.4 | 3.2 | 2.6 | 3.5 | 3.6 | 2.6 | 4.1 |

Table 5: Multiplication Timings (in $\mu$s) for different precisions with 15,000 gates in $GF(p)$-mode

As one can anticipate, stringent limitations on silicon area will have a negative impact on the time performance of the new architectures. In order to illustrate this effect, Figure 10 shows the time for various

[3]The speedup in percentage is obtained by subtracting the improved timing result from the previous timing result and then dividing it by the previous timing result.

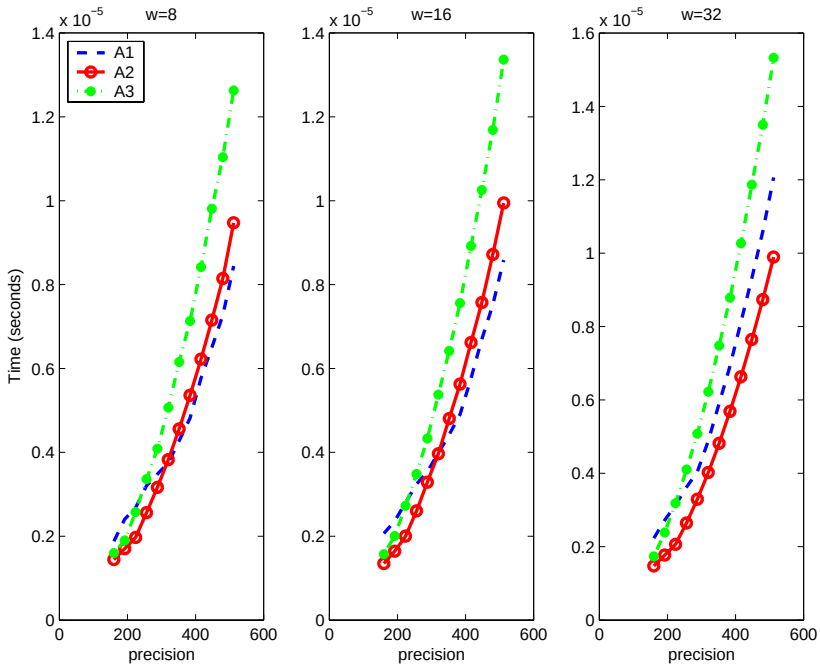


Figure 10: Multiplication Timings (in $\mu$s) for an area of 15.000 gates with $w = 8, 16$, and 32 in $GF(p)$-mode

operand precisions in a chip configuration with only 15,000 gates. The new architectures still provide a better time performance against the original design. Table 5 shows that **A2** provides a significant speedup up to 256-bits, while **A3** does up to 192-bits.

To summarize, our two new architectures provide new alternatives which facilitate a higher maximum clock frequency due to the decrease in the critical path delay and offer superior time complexity for the operand precision range of cryptographic interest. Furthermore, the dual-radix architecture, **A3**, offers a new functionality that enables to operate with radix-4 in $GF(2^n)$-mode, which halves the clock count to compute a multiplication. On the other hand, for higher precisions and for configurations with smaller area, the new architectures may result in a negative impact on the time performance in $GF(p)$-mode. However, for a very large range of precision, the cost of improving the performance for $GF(2^n)$ is a small degradation of performance for multiplication in $GF(p)$ and extra chip area.

# 5 Comparison with Previous Architectures

There is a plethora of efficient multipliers proposed for fast cryptographic computations [14, 15, 16, 17, 18, 19, 10, 12, 13, 9, 7, 8, 21, 22]. Multipliers in [18, 34, 35] offer a relatively efficient computation of multiplication for a specified precision; but they fail to scale well for other precisions. High-radix architectures such as

those in [21, 22] are specifically tailored for $GF(2^n)$ computation and it is not straightforward to apply the design approach utilized in these multipliers to unified design. Systolic-array based multipliers [7, 8, 9] usually feature high clock frequency due to low bit complexity cells composing the architecture and achieve extremely high throughput when the there are many successive multiplication operations as in the case of exponentiation operation of RSA. However, two-dimensional systolic arrays usually require large chip area and low utilization of array cells adversely affects the throughput when the multiplication operations are not successive as in the case of elliptic curve cryptography.

Our basic architecture distinguishes itself from many other architectures in the sense that it is scalable, unified, flexible, and semi-systolic that allows a low chip area with high utilization. Therefore, comparing the new architectures against any other architecture may not yield profitable insight as expected. Instead, we only compare the new architecture with two other unified and scalable designs [12, 13] proposed in the literature. The following reports on the results from comparison of different aspects of the three architectures. The comparison against [13], however, is not extensive since their architecture is based on a fast multiplication unit as opposed to carry-save adder based architecture of our work and [12].

**Time comparison:** We first compare the critical path delay and number of clocks to compute a multiplication of several precisions. While the clock counts are well documented in [12, 13], the critical path delay of their multipliers are not clearly specified. Therefore, we will estimate the critical path delays in terms of main modules in the critical path. We use **A3** (radix-(2,4)) architecture with $w = 32$ optimized for operands of up to 256-bit in the comparisons considering it is the most area consuming one among the three new architectures. In a PU of **A3** architecture, one full adder(FA) of $w$-bit is connected to a 4-to-2 multiplexer and two 2-to-1 multiplexers as seen in Figure 1. Selection logic, on the other hand, is parallel to the FA, hence not in the critical path. Unified multiplier in [12], has two layers of FA along with two 2-to-1 multiplexers and a *subtrahend generation* logic in its critical path. Assuming that the logic besides FAs in both designs is almost equivalent to each other in terms of propagation time - with a cautionary note of being imprecise -, we can formulate their critical path delay using the number of serially connected FAs and some extra delay due to multiplexers in the critical path. Therefore, having only one FA in its critical path as opposed to two FAs in [12], **A3** is expected to have less delay than that of the multiplier in [12]. Based on these assumptions, Table 6 lists critical path delay and cycle counts of the three designs.

[13], in fact, proposes three different word length (i.e. $w = 8, 16, 32$) for their multiplier based architectures. Their area complexity compares favorably with **A3** and that in [12], while its time complexity is very poor comparing to the other two architecture. As can be observed in Table 6, even the multiplier with the shortest word length ($w = 8$) has seven FAs and one $w$-bit carry propagation adder (CPA) in its critical path. In terms of both critical path delay and cycle count, [13] with $w = 8$ compares very poorly with **A3** and [12]. For larger word lengths, [13] offers very low cycle counts, while the critical path delay becomes incomparably high with carry-save based multipliers.

| Architecture type | Critical path delay | Clock Count $GF(p)$ | | | | Clock Count $GF(2^n)$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 160 | 192 | 224 | 256 | 160 | 192 | 224 | 256 |
| this work - **A3**, $w = 32$ | $\tau_{FA} + t_{extra}$ | 326 | 397 | 458 | 529 | 166 | 207 | 238 | 269 |
| [12] | $2\tau_{FA} + t_{extra}$ | 240 | 288 | 336 | 384 | 160 | 192 | 224 | 256 |
| [13], $w = 8$ | $7\tau_{FA} + \tau_{CPA}$ | 946 | 1326 | 1770 | 2278 | 882 | 1250 | 1682 | 2178 |

Table 6: Timing comparisons

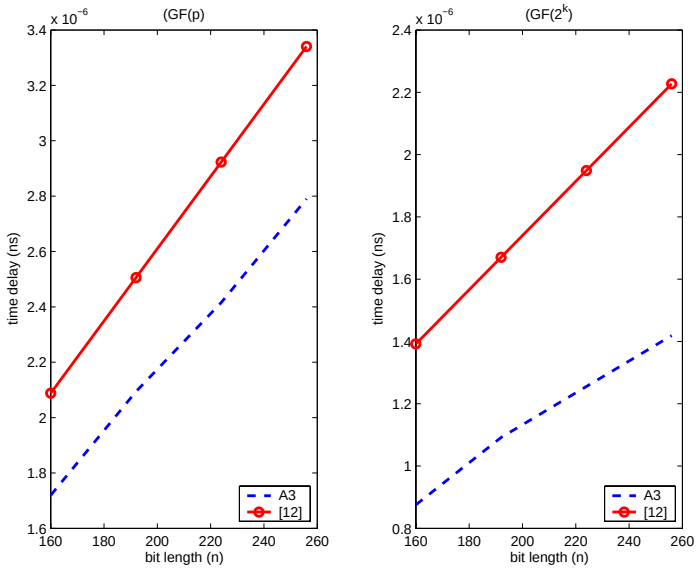


Figure 11: Time delays of **A3** and [12]

As can be observed in Table 6, [12] has longer critical path than **A3** while its cycle count is better than **A3** for $GF(p)$ (Note that the cycle counts of both designs for $GF(2^n)$ are almost the same). $t_{extra}$ in Table 6 is due to extra logic (basically multiplexers) in the critical path and its contribution to the critical path delay is found to be 54% of a FA in **A3** with $w = 32$. Thus, total critical path delay of **A3** can be given as $1.54 \cdot \tau_{FA}$. Assuming that $t_{extra}$ in [12] is equivalent to that of **A3**, the critical path delay of [12] can be approximated as $2.54 \cdot \tau_{FA}$. Taking $\tau_{FA} = 3.425$ ns in our technology, we compare the time delays of two architectures for various operand precisions in Figure 11. Clearly, **A3** architecture compares favorably with [12].

**Area and register size comparison:** Since area and register requirements are not reported in [12], we can only compare the lengths of FA layers and registers. **A3** with $w = 32$ optimized for operands up to 256 bit utilizes five PUs, hence the length of the FA layer is $5 \cdot 32 = 160$. [12], on the other hand, uses

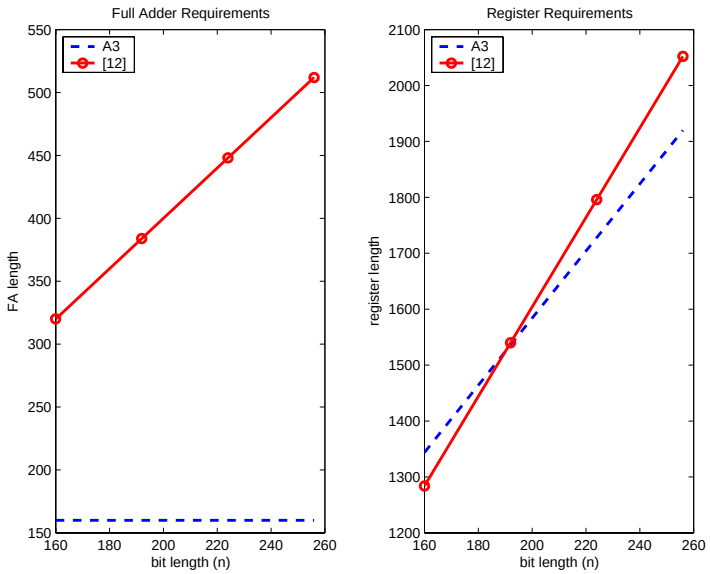


Figure 12: Area comparison of **A3** and [12]

full-precision approach, therefore, requires two FA layers of length $n$ each, where $n$ is the precision. For the lowest precision $n = 160$, the length of FA layer in [12] is twice the length of FA in **A3**. The register sizes to store all operands and temporary variables in two architectures are comparable as seen in Figure 12.

# 6 Results and Future Work

Even though very high-radix designs might introduce longer critical paths and more complex circuitry in hardware realizations, moderate range of radix values offers faster alternatives (in terms of clock cycle count to perform a multiplication) to simple radix-2 multiplier designs. While the number of clock cycles decreases in high radix designs since more multiplier bits are scanned in each clock cycle, the signal propagation time of the critical path and the silicon area increase in a similar fashion. Yet, moderately high radix multipliers may be attractive for hardware realizations when extra cost in the chip area is tolerable and very advanced VLSI technologies compensate the maximum clock frequency degradation. Note that, the new design techniques introduced in this work, can be applied to higher radix designs. For future work, it may prove to be beneficial to investigate higher radix architectures with precomputation as well as the radix-$(4, 8)$ and radix-$(8, 16)$ architectures. There are also certain optimization possibilities in the high radix members of the family such as using optimized $(4, 2)$ or $(5, 2)$ dual-field adders instead of using two or three layers of $(3, 2)$ dual-field adders. This can be easily seen from Appendix B, where the architecture and selection logic of radix-(4,8) multiplier are given.

Using the design methodology proposed in [10], we presented two new unified multiplier architectures for binary extension and prime fields. The first 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 second architecture (*dual-radix*) facilitates faster computation of multiplication in $GF(2^n)$-mode than $GF(p)$-mode. The dual-radix architecture also utilizes the precomputation technique. 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 other multiplier architectures such as [12] and its feasibility is discussed. 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, while it suffers from a slight performance deterioration when compared against the architecture with precomputation in $GF(p)$-mode, 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] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, 1993.

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

[7] C. D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–378, March 1993.

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

[9] W. L. Freking and K. K. Parhi. Performance-scalable array architectures for modular multiplication. *Journal of VLSI Signal Processing*, (31):101–106, 2002.

[10] 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.

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

[12] 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.

[13] A. Satoh and K. Takano. A scalable dual-field elliptic curve cryptographic processor. *IEEE Transactions on Computers, Special Issue on Cryptographic Hardware and Embedded Systems*, (4):449–460, April 2003.

[14] 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.

[15] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In S. Knowles and W. H. McAllister, editors, *Proceedings, 12th Symposium on Computer Arithmetic*, pages 193–199, Bath, England, July 19–21 1995. IEEE Computer Society Press, Los Alamitos, CA.

[16] S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, June 1993.

[17] 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.

[18] A. Royo, J. Moran, and J. C. Lopez. Design and implementation of a coprocessor for cryptography applications. In *European Design and Test Conference*, pages 213–217, Paris, France, March 17-20 1997.

[19] A. F. Tenca, G. Todorov, and Ç. K. Koç. High-radix design of a scalable modular multiplier. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001*, Lecture Notes in Computer Science No. 2162, pages 189–205. Springer, Berlin, Germany, 2001.

[20] 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.

[21] L. Song and K. K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *Journal of VLSI Signal Processing*, (19):149–166, 1998.

[22] M. C. Mekhallalati, A.S. Ashur, and M. K. Ibrahim. Novel radix finite field multiplier for $gf(2^m)$. *Journal of VLSI Signal Processing*, (15):233–245, 1997.

[23] A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science No. 1717, pages 94–108. Springer, Berlin, Germany, 1999.

[24] IEEE. P1363: Standard specifications for public-key cryptography. 2000.

[25] S. E. Eldridge. An faster modular multiplication algorithm. *International Journal of Comput. Math*, 40:63–68, 1991.

[26] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[27] J.-H.Oh and S.-J.Moon. Modular multiplication method. *IEE Proceedings*, 145(4):317–318, July 1998.

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

[29] G. Hachez and Jean-Jacques Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In *Cryptographic Hardware and Embedded Sytems*, Lecture Notes in Computer Science, No. 1965, pages 293–301. Springer-Verlag, Berlin, 2000.

[30] E. Savaş. *Implementation Aspects of Elliptic Curve Cryptography*. PhD thesis, Department of Electrical and Computer Engineering, Oregon State University, June 2000.

[31] G. Todorov. Asic design, implementation and analysis of a scalable high-radix Montgomery multiplier. Master's thesis, Department of Electrical and Computer Engineering, Oregon State University, December 2000.

[32] ASIC design kit. Mentor Graphics Co.

[33] B. Kurniawan. Asic design and implementation of a parallel exponentiation algorithm using optimized sacalable Montgomery multipliers. Master's thesis, Department of Electrical and Computer Engineering, Oregon State University, 2002.

[34] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

[35] D. Naccache and D. M'Raïhi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, June 1996.

# A Derivation of Formulae for Selection logic

Selection logic calculates the coefficients of $b$ and $p$ in the following formula

$$c = (c + a_i \cdot b + q \cdot p) >> 1$$

where $q = (c_0 + a_i \cdot b_0) \cdot p'_0$ and q, $a_i$, $b_0$, $c_0$, and $p'_0$ are radix-4 digits. In $GF(p)$-mode, we have

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

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

where $p'_{0,0} = p_{0,0} = 1$.

In $GF(2^n)$-mode, we need to compute $m_{00}$, $m_{01}$, $m_{10}$, and $m_{11}$. $m_{00}$ is the same as in the $GF(p)$-mode. $m_{10}$ must be forced to 0 in $GF(p)$-mode, thus we have $m_{10} = a_{i,1} \cdot \overline{FSEL}$. $m_{01}$ and $m_{11}$ are determined by $q$. In order to have $q$ we need to calculate $p'_0$ first. We know that $p_0 \cdot p'_0 \equiv 1 \pmod{x^2}$. Therefore,

$$(p'_{0,1} \cdot x + p'_{0,1}) \cdot (p_{0,1} \cdot x + p_{0,1}) \equiv (p'_{0,1} + p'_{0,1} \cdot p'_{0,1}) \cdot x + p'_{0,0} \equiv 1 \pmod{x^2}$$

Consequently, $p'_{0,0} = 1$ and $p'_{0,1} = p_{0,1}$. Also,

$$a_i \cdot b_0 \equiv (a_{i,1} \cdot x + a_{i,0}) \cdot (b_{0,1} \cdot x + b_{0,0}) \equiv (a_{i,1} \cdot b_{0,0} + a_{i,0} \cdot b_{0,1}) \cdot x + a_{i,0} \cdot b_{0,0} \pmod{x^2}$$

Therefore,

$$(c_0 + a_i b_0) \cdot p'_0 \equiv [(cs_{0,1} + a_{i,1} \cdot b_{0,0} + a_{i,0} \cdot b_{0,1}) \cdot x + (cs_{0,0} + a_{i,0} \cdot b_{0,0})](p_{0,1} \cdot x + 1) \pmod{x^2}$$

$$\equiv [cs_{0,1} + a_{i,1} \cdot b_{0,0} + a_{i,0} \cdot b_{0,1} + (cs_{0,0} + a_{i,0} \cdot b_{0,0})p_{0,1}] \cdot x + cs_{0,0} + a_{i,0} \cdot b_{0,0}$$

Finally we obtain,

$$m_{01} = cs_{0,0} \oplus a_{i,0} \cdot b_{0,0}$$

and

$$m_{11} = cs_{0,1} \oplus a_{i,1} \cdot b_{0,0} \oplus a_{i,0} \cdot b_{0,1} + (cs_{0,0} \oplus a_{i,0} \cdot b_{0,0})p_{0,1}.$$

Since $cc_{0,0}$ is always 0 in $GF(2^n)$-mode, we can use

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

for both mode. In addition, $m_{11}$ is forced to 0 in $GF(p)$-mode, hence

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

# B Radix-(4,8) Multiplier and Its Selection Logic

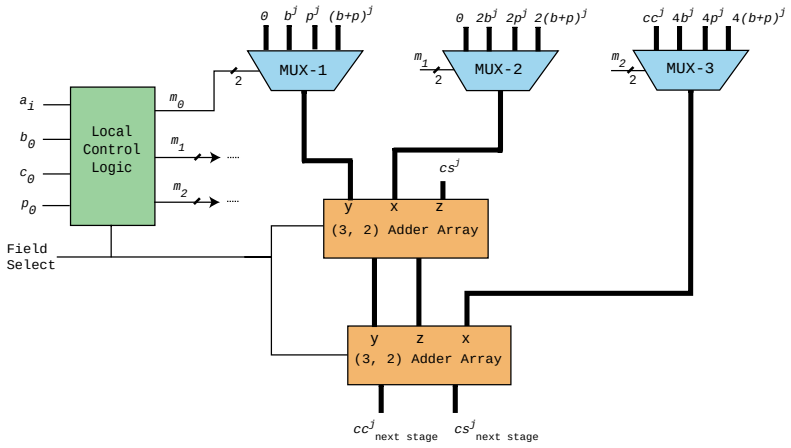


Figure 13: Processing unit of dual-radix architecture with radix-4 for $GF(p)$ and radix-8 for $GF(2^n)$

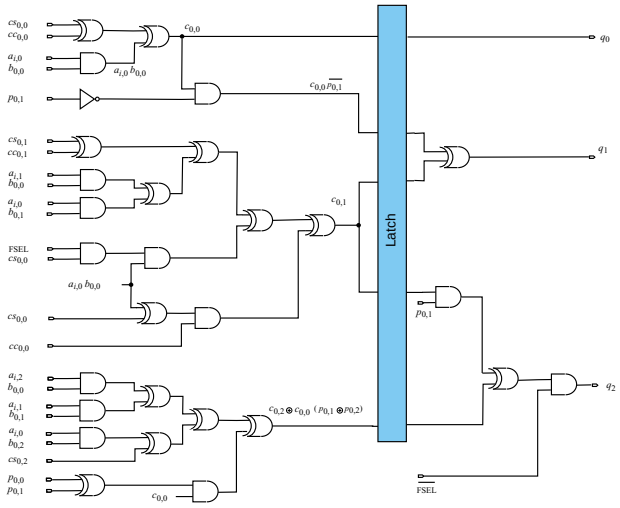


Figure 14: Selection logic for radix-(4,8) multiplier

**Set of Tables**

- **Table 1:** Selection Inputs to MUX-0 and MUX-1
- **Table 2:** Relative increase in design area
- **Table 3:** Speedup in delay
- **Table 4:** Multiplication Timings (in $\mu$s) for different precisions with 30,000 gates in $GF(p)$-mode
- **Table 5:** Multiplication Timings (in $\mu$s) for different precisions with 15,000 gates in $GF(p)$-mode
- **Table 6:** Timing comparisons

**Set of Figures**

- **Figure 1:** Processing unit of dual-radix architecture with radix-2 for $GF(p)$ and radix-4 for $GF(2^n)$
- **Figure 2:** Execution graph of Montgomery multiplication algorithm [23]
- **Figure 3:** Pipeline organization with two processing units
- **Figure 4:** Dual Field Adder Array for radix-(2,4) Unified Multiplier
- **Figure 5:** Dual-Field Adder Cell
- **Figure 6:** Shift and Alignment Layer
- **Figure 7:** Selection logic for radix-(2,4) multiplier
- **Figure 8:** Implementation results: Critical path delay and area
- **Figure 9:** Multiplication Timings (in $\mu$s) for an area of 30.000 gates with $w = 8, 16$, and 32 in $GF(p)$-mode
- **Figure 10:** Multiplication Timings (in $\mu$s) for an area of 15.000 gates with $w = 8, 16$, and 32 in $GF(p)$-mode
- **Figure 11:** Time delays of **A3** and [12]
- **Figure 12:** Area comparison of **A3** and [12]
- **Figure 13:** Processing unit of dual-radix architecture with radix-4 for $GF(p)$ and radix-8 for $GF(2^n)$
- **Figure 14:** Selection logic for radix-(4,8) multiplier