AN ABSTRACT OF THE THESIS OF

Lo'ai Ali Tawalbeh for the degree of Doctor of Philosophy in Electrical & Computer Engineering presented on October 28, 2004.

Title: A Novel Unified Algorithm and Hardware Architecture for Integrated Modular Division and Multiplication in $GF(p)$ and $GF(2^n)$ Suitable for Public-Key Cryptography.

Abstract approved: ______________________________

Çetin Kaya Koç

The spread of the internet and communications techniques increases the necessity for security in applications that involves sharing or exchange of secret or private information. Public-key cryptography is widely used in establishing secure communication channels between the users on the Internet, for E-commerce transactions, and in network security protocols. Public-key cryptography relies on algorithms from computer arithmetic, number theory and algebra. The modular arithmetic operations, modular division, and modular multiplication over finite fields ($GF(p)$ and $GF(2^n)$) are extensively used in many public-key cryptosystems, such as RSA, ElGamal cryptosystem, Diffie-Hellman key exchange algorithm, elliptic curve cryptography (ECC), and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm. In our research, we have mainly concentrated on hardware realization of the ECC since it seems to provide similar amount of security using smaller key size.

The modular multiplication operation with a large modulus is very important in many public-key cryptosystems. One of the most efficient ways to compute modular multiplication is the Montgomery algorithm. Many efficient Montgomery multiplier designs were proposed up to now. On the other hand, computing modular division (inverse) is a time-consuming process and cannot be avoided completely. It was claimed that a gain in performance can be obtained when implementing the division (inverse) in hardware.

In this work, we propose, with a mathematical proof, an efficient unified division algorithm to compute the modular division operation in $GF(p)$ and $GF(2^n)$. The algorithm uses a counter to keep track of the difference between two field elements and this way eliminates the need for comparisons which are usually expensive and time-consuming. An hardware architecture implementing the algorithm is also proposed.

The unified division algorithm is integrated with a unified Montgomery multiplication algorithm to obtain a novel Unified Division/Multiplication Algorithm (UDMA). The UDMA computes division (inverse) and multiplication in a very efficient way in both $GF(p)$ and $GF(2^n)$ fields. Also, we propose a unified hardware architecture that efficiently supports all operations in the UDMA and uses carry-save unified adders for reduced critical path delay, making the proposed architecture faster than other previously proposed designs.

Experimental results obtained by synthesizing the hardware design for AMI $0.5\mu m$ CMOS technology and FPGA $VertixII$ chip ($xc2vp50-7ff148$ technology) are shown and compared with other proposed dividers and multipliers.

A Novel Unified Algorithm and Hardware Architecture for Integrated Modular Division and Multiplication in $GF(p)$ and $GF(2^n)$ Suitable for Public-Key Cryptography

by

Lo'ai Ali Tawalbeh

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented October 28, 2004
Commencement June 2005

ACKNOWLEDGMENTS

First of all, I am grateful to the GOD, most mighty, most merciful, who created me and gave me the ability to think and search for the truth.

I would like to thank my major advisor, prof. Çetin Kaya Koç, for his help, support, encouragement, and valuable directions towards accomplishing this work. Also, I want to thank Dr. Alexandre F. Tenca who was my advisor during my Master study (September 2001- October 2002), and part of my PhD study. Dr. Tenca's discussions, directions, reviews and comments helped me much in conducting this research and writing conference and journal publications which formed as the bases of this thesis. I also thank the other members of my graduate committee for their valuable input.

I am thankful to my research colleagues in the Information Security Laboratory of Oregon State University. In particular, I acknowledge Mohammad Sinky, Song Park, Ajay Shantilal, Minho Kim, Colin Van Dyke, Onur Aciiçmez and Gökay Saldamli, for providing valuable feedback during different research phases.

The financial support for my PhD education is from a scholarship provided by Jordan University of Science and Technology (JUST)- Jordan, and NSF CAREER grant CCR-0093434- "Computer Arithmetic Algorithms and Scalable Hardware Designs for Cryptographic Applications".

Finally, this work will not have been possible without the love and moral support of my father (Ali Tawalbeh), my patient, great and wonderful mother, my brothers (Qosai, Ahmad and Mohammad), my sisters, and my aunts (Um Anas and Um Ibrahim). I am deeply thankful to all of them for their support and warm feelings during my study years overseas.

Lo'ai Ali Tawalbeh

Corvallis, Oregon, USA

October , 2004

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF APPENDIX TABLES

To my mother and my father, with love ..

A NOVEL UNIFIED ALGORITHM AND HARDWARE ARCHITECTURE FOR INTEGRATED MODULAR DIVISION AND MULTIPLICATION IN $GF(P)$ AND $GF(2^N)$ SUITABLE FOR PUBLIC-KEY CRYPTOGRAPHY

1. INTRODUCTION.

The spread of the internet and communications techniques increases the necessity for security in applications that involves sharing or exchange of secret or private information. Public-key cryptography is widely used in establishing and verifying communications between the users on the web, E-commerce transactions, and network security protocols.

Among the modular arithmetic operations, modular division (notice that division includes computing the inverse) and multiplication over finite fields ($GF(p)$ and $GF(2^n)$) are extensively used in many cryptographic applications, such as ElGamal cryptosystem [1], Diffie-Hellman [2] key exchange algorithm, RSA [3], elliptic curve cryptography, and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm[4].

In this work, we: first, propose an efficient unified modular division algorithm to compute modular division in $GF(p)$ and $GF(2^n)$ and its hardware architecture. We also provide a mathematical proof for the algorithm. The algorithm uses a counter to keep track of the difference between two field elements and this way eliminates need for comparisons which are usually expensive and time-consuming. The hardware architecture that implements the algorithm is also proposed.

Second, knowing that the Montgomery multiplication algorithm [5] is one of the most efficient ways to compute modular multiplication, we modify the unified Montgomery multiplication algorithm proposed by other members of our research group in [6] to have a control flow similar to the proposed division algorithm.

After that, we integrate the proposed unified division algorithm in this work with the unified Montgomery multiplication algorithm in [6] to get a novel Unified modular Division/Multiplication Algorithm (UDMA). The UDMA computes division (inverse) and multiplication in a very efficient way in both $GF(p)$ and $GF(2^n)$ fields. Also, we propose the hardware architecture of the algorithm which efficiently supports all the operations in the UDMA, and uses carry-save unified adders for reduced critical path delay, making the proposed architecture faster than other previously proposed designs. The added scalability feature of the proposed divider/multiplier allows a fixed-area datapath to handle operands of any size. Also, the word size of the datapath can be adjusted to meet the area and performance requirements.

Experimental results obtained by synthesizing the hardware design for AMI $0.5\mu m$ CMOS technology and FPGA $VertixII$ chip ($xc2vp50 - 7ff148$ technology) are shown and compared with other proposed dividers and multipliers.

Next section of this Chapter presents the motivation behind conducting this research. Section 2 shows previous work related to computing modular division (inverse) and Montgomery multiplication in hardware. Section 3 describes Montgomery multiplication, and presents a modified version of the unified Montgomery multiplication algorithm proposed in [6]. The organization of this thesis is presented in Section 4.

1.1. Motivation

Modular arithmetic operations such as division and multiplication over finite fields ($GF(p)$ and $GF(2^n)$), are widely used in several cryptographic applications. The modular multiplication operation with a large modulus is very important in many public-key cryptosystems such as the RSA algorithm [3].

On the other hand, modular division and modular inverse are time consuming operations and they cannot be avoided completely in practical applications. For instance, they are used in the ElGamal [1] public-key cryptosystem and the Diffie-Hellman key exchange method [2]. Modular inversion is also considered as an essential operation in the Elliptic Curve Cryptography (ECC) [4, 7, 8]. This research is targeted mainly toward the ECC utilization because of its promise to replace several older cryptographic systems [9, 10]. Researchers have claimed that a gain in performance can be obtained when modular division and inversion are implemented in hardware [9, 11]. Also, for cryptographic applications, it is more secure to implement all the computations in hardware than performing some computations in software with others processed in hardware. This is because software implementations are supported by operating systems, which can be interrupted and trespassed by intruders, compromising the application security. On the other hand, such a security threat is not so easily attained in hardware implementations [9].

In general, there is an increasing demand for dedicated hardware to accelerate the huge amount of computations required by public-key cryptographic algorithms. An algorithm and hardware implementation that is able to compute modular division and multiplication in both $GF(p)$ and $GF(2^n)$ is definitely advantageous and has great importance to systems that need to quickly switch between these fields, such as network servers. The scalability feature of a hardware module is very useful and allows the users to use a fixed-area module to handle operands of any size. The word size of the module can be adjusted to meet the best area/performance requirements.

1.2. Literature Review

The Extended Euclidean Algorithm (EEA) is an efficient way to compute modular division [12, 13]. There are several proposed design in the literature that computes modular division (inverse) [9, 11, 14, 15, 16, 17, 18, 19, 20, 21] based on the EEA or one of its modifications [22]. Most of the proposed designs compute the inverse in the binary extension fields – $GF(2^n)$ [14, 17, 18, 19, 20, 21, 23]. The designs proposed in

The Design	Area Complexity	Time Complexity
Guo and Wang [17]	$O(nlogn)$	$O(n)$
Choudhury and Barua [18]	$O(n)$	$O(n^2)$
Kovac, Ranganathan and Varanasi [25]	$O(n^3)$	O(1)
Guo and Wang [24]	$O(n)$	$O(n)$
Daneshbeh and Hasan [14]	$O(n)$	$O(n)$

TABLE 1.1: Area and time complexity of some inversion designs

[20, 23] suffer from signal broadcasting problem which should be avoided in high-speed VLSI circuits [12].

Other designs are based on the concept of systolic array structures [14, 17, 18, 24, 25]. A systolic array is an arrangement of interconnected logic cells in array where data flow synchronously between the adjacent cells. Systolic arrays are simple and has regular communications and control structures which make it suitable for VLSI implementations. But, on the other hand, a huge amount of hardware area is needed in order to gain computation speed [26]. Table 1.1 shows the area and time complexities of some inversion designs.

A VLSI algorithm for modular division based on the Binary GCD algorithm was proposed in [27]. The algorithm is based on the plus-minus algorithm presented in [28] which is a modification of the binary method for calculating the Greatest Common Divisor (GCD). The redundant binary representation is used to avoid carry propagation. The same author with cooperation with another researcher proposed in [29] a VLSI algorithm to compute division and multiplication in $GF(p)$ only, which uses the same algorithm proposed in [27] for division. The algorithm was implemented in a linear array structure that uses signed digit representation and performs n-bit modular multiplication in $\lfloor \frac{2(2n+3)}{3} \rfloor + 3$ clock cycles and modular division in $2n + 5$ clock cycles.

The public-key processor presented in [16] implements operations required for Elliptic Curve Cryptography (ECC) including modular inverse in $GF(2^n)$. Its has a reconfigurable datapath. The processor hardware is more energy efficient and faster than software implementations. But, on the other hand, it has a very large area.

Another work in [15] presents a simple dual-field arithmetic unit, however, an unified algorithm for modular inverse/division was not shown. The unit performs one addition in each clock cycle, and the redundant adder in the architecture is used to convert from the carry-save form to the non-redundant representation, significantly increasing the number of clock cycles.

The Montgomery multiplication algorithm proposed in [5] is considered a very efficient way to compute modular multiplication. An implementation of Montgomery multiplication should consider the tradeoff between chip area and computational speed [30, 31].

A flexible multiplier can be integrated into a system as an autonomous co-processor attached to the system bus [6, 32]. Also, the multiplier can be integrated as a functional unit to the main CPU. With the idea of implementing more cryptographic operations in hardware, this approach is becoming increasingly attractive [33, 34].

A single chip, 1024-bit RSA implementation is shown in [35]. The multiplication part is implemented as an array multiplier. This approach for multiplication requires multiple clock cycles to complete. Another approach to perform modular multiplication, is to use a core with a small bit size and reuse it with bit portions of the operands [30]. It is shown in [36] that limiting the size of the computing unit has certain advantages.

Implementing the multiplier using reconfigurable hardware provides the means of solving problems for both high-precision and variable-precision computation [30]. The main candidates for flexible hardware are FPGAs [32, 37]. Thus, in this work, we are implementing the proposed hardware architecture in FPGAs in addition to the ASIC implementation.

A very good representative of Montgomery multiplier implementation is the unified multiplier architecture for both finite fields, $GF(p)$ and $GF(2^m)$, is presented in [6]. It shows that a Montgomery multiplication module can operate in both fields without significant increases in the design area compared to a multiplier that works on $GF(p)$ only.

1.3. Montgomery Modular Multiplication

In this Section, we explain briefly the Montgomery multiplication algorithm. We rewrote the algorithm and used the same notation as in the unified modular division algorithm proposed in this work to emphasize the similitude between the two algorithms. The similarities between the two algorithms are used to get a novel Unified modular Division and Multiplication Algorithm (UDMA) which is described in Chapter 3.

The Montgomery multiplication algorithm generates the product of two n-bit integers Y (multiplier) and X (multiplicand) in modulo p according to the following expression:

$$MM(Y, X) = YXr^{-1} \; mod \; p$$

where $r = 2^n$. p is chosen such that the greatest common divisor of r and p is one (gcd(r,p) = 1), which indicates that r and p should be relatively prime. This condition is easily achieved by choosing p as an odd integer, since $r = 2^n$ is an even number. We usually have $2^{n-1} < p < 2^n$. The Montgomery image of an integer can be obtained by multiplying it by the constant r and taking it modulo p: $\bar{a} = ar \; mod \; M$.

The Montgomery multiplication over the images $\bar{a}$ and $\bar{b}$ results in:

$$\bar{c} = cr \; mod \; M = MM(\bar{a}, \bar{b}) = abr \; mod \; M$$

which corresponds to the image of $c = ab \; mod \; M$, the modular product of a and b.

Figure 1.1 shows the transformation between the integers and their images performed using MM. This process can be explained as follows:

- to transform an integer a to its image $\bar{a}$, we do: $\bar{a} = MM(a, r^2) = ar^2r^{-1} \; modM = ar \; mod \; M$.

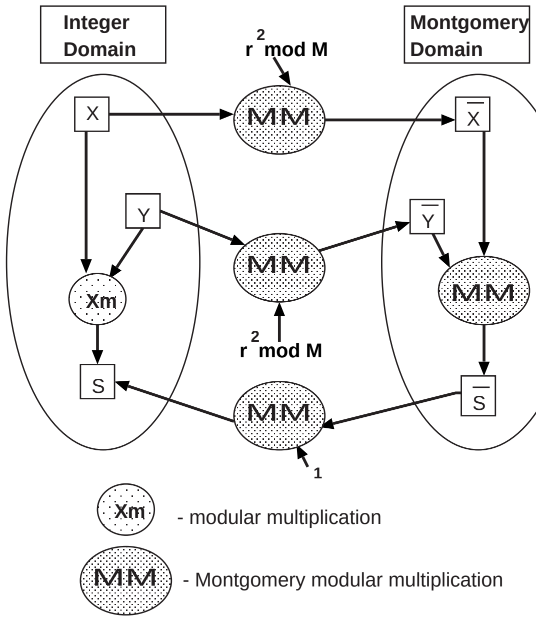


FIGURE 1.1: Modular multiplication using MM.

- to transform from an image $\bar{a}$ to the integer a, we compute: $a = MM(\bar{a}, 1) = arr^{-1}\ mod\ M = a\ mod\ M$.

Observe that the constant $r^2\ mod\ M$ is pre-computed and used in the process as shown in Figure 1.1.

The radix-2 unified Montgomery multiplication algorithm is shown in Figure 1.2 [6].

The Montgomery multiplication algorithm performs n iterations. In each iteration one bit of the multiplier is tested (we are considering $C = Y$ at the beginning). If $c_0 = 1$, the C variable is shifted one bit to the right, and one multiple of the multiplicand X is added to the partial product U. If $c_0 = 0$, only a shift right is done on C and the next bit is tested. Note that U is always bounded by $2p$ through all iterations. Therefore, the last correction step (modular reduction) assures that the output is correctly presented in modulo p. In order to use the result of one multiplication as the input to another one without modular reduction, we need to use two extra bits of precision ($n + 2$ bits) [38, 33].

1.4. Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents A new unified division algorithm and it's hardware architecture that computes modular division (inverse) in $GF(p)$ and $GF(2^n)$. A novel Unified Division and Multiplication Algorithm (UDMA) which is able to compute both modular division (inverse) and Montgomery multiplication in $GF(p)$ and $GF(2^n)$ fields, is presented in Chapter 3. We propose a scalable and unified modular divider/multiplier hardware architecture for the UDMA in Chapter 4. The architecture is implemented in ASIC and Field Programmable Gate Arrays (FPGAs). The experimental results are presented and compared with previous designs in Chapter 5. The conclusions and future work are shown in Chapter 6.

[Montgomery Modular Multiplication in $GF(p)$ and $GF(2^n)$ fields]

Inputs: $0 \leq X, Y < p$, $2^{n-1} < p < 2^n$, $Field$, n

Output: $Z = XY2^{-n} \bmod p$ when $Field = GF(p)$, $Z(x) = X(x)Y(x)2^{-n} \bmod p(x)$ when $Field = GF(2^n)$

Algorithm:

$C = Y$, $U = 0$, $W = X$

```
FOR $i = 1$ To $n$
  IF $c_0 = 0$ THEN
    $C := C >> 1$
  ELSE                /* $c_0 = 1$ */
    $C := C >> 1$,    $U := (U + W)$
  END IF;
  $U := (U + u_0 * p) >> 1$
END FOR;
IF $U \geq p$ THEN $Z := U - p$
ELSE $Z := U$
END IF;
```

FIGURE 1.2: Modified version of the unified Montgomery multiplication algorithm presented in [6].

2. NEW UNIFIED MODULAR DIVISION ALGORITHM AND HARDWARE ARCHITECTURE IN $GF(P)$ AND $GF(2^N)$.

This Chapter presents an algorithm to compute modular division in both $GF(p)$ and $GF(2^n)$ fields (also called *unified*). Since computing division includes computing the inverse, we will use the term division because it is more general. In the following Section, we review some mathematical concepts and introduce the notation to be used in this Chapter. In Section 2.2, we present the Unified Modular Division (UMD) algorithm with its mathematical proof. The hardware architecture that implements the UMD algorithm is described in Section 2.3.

2.1. Mathematical Concepts and Notation

The binary extension field element, $Y(x) \in GF(2^n)$, is a non-zero polynomial of degree less than n, when the polynomial basis is used to represent the field elements (which is the case in this thesis). Each element has coefficients in $GF(2)$, which are represented by the values $\{0, 1\}$. On the other hand, the elements in the prime field $GF(p)$ are integers in the range $\{0, ..., p-1\}$ where p is a n-bit prime modulus in the range $2^{n-1} < p < 2^n$. Bit vectors are used to represent the elements in both fields as follows:

$$GF(2^n) : Y(x) = \sum_{i=0}^{n-1} y_i * x^i$$

$$GF(p) : Y = \sum_{i=0}^{n-1} y_i * 2^i$$

where $y_i \in \{0, 1\}$ in both cases. The polynomial $Y(x)$ is denoted as Y in the algorithm description for simplicity.

The addition operation of the elements is different in each field. Addition of two polynomials in $GF(2^n)$ is done by a bitwise logic exclusive OR operation (a xor $b = a \oplus b$

$= a'b + ab'$) between the two bit vectors being added. In other words the additions in $GF(2^n)$ are done modulo 2 [15], as shown in the following equation:

$$Y(x) + W(x) = \sum_{i=0}^{n-1} y_i * x^i + \sum_{i=0}^{n-1} w_i * x^i = \sum_{i=0}^{n-1} (y_i \ xor \ w_i) x^i$$

Subtraction and addition in $GF(2^n)$ are equivalent. Intermediate results of operations in $GF(2^n)$ that are represented by polynomials of degree greater or equal to n are reduced using a field polynomial $p(x)$ of degree n (irreducible polynomial) [39].

Moreover, the addition of two elements Y and W in $GF(p)$ is done as a conventional integer addition. The propagation of carries in this case depends on the use of redundant or non-redundant representation of elements. Carry-Save (CS) representation is used in this work. Modular reduction is required when the sum exceeds the value of p to keep the result in the set $\{0, ..., p-1\}$.

2.2. Unified Modular Division Algorithm (UMD)

The Unified Modular Division (UMD) algorithm is shown in Figure 2.1. The algorithm is based on the Extended Binary GCD algorithm [27]. Most of the modular division (inverse) algorithms [11, 16, 15, 14, 21, 17] have integer and polynomial degree comparisons as part of their control flow. Differently from them, the UMD algorithm uses a counter variable to keep track of the difference between field elements, and this way, eliminates the need for comparisons and complex tests which are usually expensive and time consuming. Using the counter results in a less complex iterations [40], and it can be implemented using fast up/down counters as the ones described in [41].

The UMD algorithm computes the modular division in $GF(2^n)$ when $Field = GF(2^n)$ ($Z(x) = \frac{X(x)}{Y(x)} \ mod \ p(x)$), and in $GF(p)$ when $Field = GF(p)$. In both cases, $Y \neq 0$. If X is set to one, the UMD algorithm computes the modular inverse. We must say that the operations on the control variable δ are always integer operations regardless of what is the specified field. On the other hand, specifying a field forces all the additions/subtractions to be done in this field. Swap of values between two variables is indicated by the symbol $\Leftrightarrow$. The notation for the least-significant bits of C and U is c_0 and u_0, respectively. The symbol $>>$ indicates right shift by one bit (divide by 2 in $GF(p)$ or by x in $GF(2^n)$). Notice that inputs to the UMD algorithm (X ,Y) are bit vectors that represent elements in $GF(p)$ and $GF(2^n)$.

The algorithm is based on the following facts to compute the division in $GF(p)$ [12, 27, 40]: If C is even and D is odd, then $gcd(C, D) = gcd(C/2, D)$. If C and D are both odd, then 4 divides either $C + D$ or $C - D$. If the first is true, then $gcd(C, D) = gcd((C + D)/2, D) = gcd((C + D)/4, D)$, and $|(C + D)/4| \leq max(|C/2|, |D/2|)$. If 4 divides $(C - D)$ then $gcd(C, D) = gcd((C - D)/2, D) = gcd((C - D)/4, D)$, and $|(C - D)/4| \leq max(|C/2|, |D/2|)$. In the algorithm, these operations are performed by $C := (C + kD)/2$ in one iteration and $C := C/2$ in the following iteration. In any case, since the result of these operations is stored back into C, when $C > D$, the size of the bit vector C is reduced by 1 bit. If $C < D$, the size of the bit vector may not be reduced, and the counter is used to limit the number of iterations when the algorithm stays in this situation, forcing the swap of variables to the condition $C > D$ which is required for convergence.

In $GF(2^n)$ one can show that $gcd(C, D) = gcd((C+D)/x, D)$ (addition is the same as subtraction in this field) but the divisibility by 4 can not be enforced. Therefore, when $Field = 0$, the test $(C + D) \bmod 4 \neq 0$ is not applicable. When $deg(C) > deg(D)$, we can see that $deg((C + D)/x) \leq deg(C/x) = deg(C) - 1$, and therefore, C will have its degree reduced in each iteration.

The combination of the two expressions: $U = (U + kW)$ and $U := (U + u_0 p) >> 1$, where u_0 is the least-significant bit of U, implements the operation $U := (U + kW) >> 1 \bmod p$. This way the modular reduction is done by a simple conditional addition of the modulus in $GF(p)$ or irreducible polynomial in $GF(2^n)$.

The UMD algorithm can be used to compute the inverse in Montgomery domain by the use of pre-computed constants. Considering the Montgomery images r (corresponds to 1) and Y as inputs, where $r = 2^n$, we get $Z = \frac{r}{Y} \pmod{p} = Y^{-1}r \pmod{p}$ which is

Function: Modular Division in $GF(p)$ and $GF(2^n)$ fields

Inputs: $0 \leq X < p$, $0 < Y < p$, $2^{n-1} < p < 2^n$, $Field$

Output: $Z = \frac{X}{Y} \mod p$ when $Field = GF(p)$, $Z(x) = \frac{X(x)}{Y(x)} \mod p(x)$ when $Field = GF(2^n)$

Algorithm:

$C = Y$, $U = X$, $D = p$, $W = 0$, $\delta = 0$

```
WHILE $C \neq 0$
  IF $c_0 = 0$ THEN
    $C := C >> 1$
    $\delta := \delta - 1$          /* Integer Operation */
  ELSE
    IF $\delta < 0$ THEN $C \Leftrightarrow D$, $U \Leftrightarrow W$, $\delta := -\delta$
    END IF;
    $k := 1$
    IF($(C + D)$ $mod$ $4 \neq 0$ AND $Field = GF(p)$) THEN $k := -1$
    ELSE    $\delta := \delta - 1$
    END IF;
    $C := (C + k * D) >> 1$, $U := (U + k * W)$;
  END IF;
  $U := (U + u_0 * p) >> 1$
END WHILE;
IF $D = 1$ THEN $Z := W$
ELSE $Z := p - W$
END IF;
```

FIGURE 2.1: Unified Modular Division Algorithm (UMD)

the inverse in the Montgomery domain. But, if we use r and Yr as inputs, the algorithm computes the inverse in the integer domain ($Z = Y^{-1} \pmod{p}$).

2.2.1. Numerical Example

Table 2.1 shows a numerical example of modular division using the UMD algorithm. The algorithm computes $\frac{213}{108}$ *mod* 251 in $GF(p)$. The Table shows the values of the variables: $delta, C, D, U$, and W. The first column shows the operation performed by the division algorithm. For instance, $C \;\; >> \;\; 1$ indicates that the $THEN$ portion was executed (C is even), and so, C was shifted and δ was decremented. On the other hand, the expression $C := (C \pm D) >> 1$ indicates that the $ELSE$ portion was executed (C is odd), and so, $U := (U \pm W)$ was also performed. The operation *swap* means that $C \;\Leftrightarrow\; D$ and $U \;\Leftrightarrow\; W$, and the sign of δ was flipped ($\delta = -\delta$). Notice that the modular reduction step ($U := (U + u_0 * p) >> 1$) is performed every iteration regardless of C being odd or even.

The computations are done when $C = 0$ and the result is $W = 246$ as can be seen form the Table.

2.2.2. Mathematical Proof

The algorithm proposed in [27] computes division in $GF(p)$ only. It applies the Plus-Minus algorithm of Brent and Kung presented in [28]. This last, is a modification of the standard Binary algorithm to compute the GCD [42]. We adopted similar approach used in [27] with many modifications including extending the algorithm to compute division in both $GF(p)$ and $GF(2^n)$ fields in a more efficient way, and simpler flow to be more suitable for hardware implementation.

In both possible cases of fields, the main justification of the success of the algorithm is that throughout the computation, one has the two congruences:

Operations	δ	C	D	U	W
Initialization	0	108	251	213	0
$C >> 1$	-1	54	251	232	0
$C >> 1$	-2	27	251	116	0
$swap$,$C := (C - D) >> 1$	2	112	27	-58	116
$C >> 1$	1	56	27	-29	116
$C >> 1$	0	28	27	111	116
$C >> 1$	-1	14	27	181	116
$C >> 1$	-2	7	27	216	116
$swap$,$C := (C - D) >> 1$	2	10	7	-50	216
$C >> 1$	1	5	7	-25	216
$C := (C + D) >> 1$	1	6	7	221	216
$C >> 1$	0	3	7	236	216
$C := (C - D) >> 1$	0	-2	7	10	216
$C >> 1$	-1	-1	7	5	216
$swap$,$C := (C - D) >> 1$	1	4	-1	231	5
$C >> 1$	0	2	-1	241	5
$C >> 1$	-1	1	-1	246	5
$swap$,$C := (C + D) >> 1$	1	**0**	1	0	**246**

TABLE 2.1: A modular division by the UMD algorithm in $GF(p)$

$$CX \equiv UY \bmod p$$

$$DX \equiv WY \bmod p \;;$$

Indeed, this is clearly true upon initialization: $C = Y$; $U = X$ and $D = p$, $W = 0$. Thereafter, we either: swap ordered pairs (C,U) with (D,W); or, replace (C,U) by $(\,(C + kD)\xi^{-1},\,(U + kW + v_0 p)\xi^{-1}\,)$, with a shared value of k and with $v_0 \in \{0,1\}$, and ξ^{-1} is either 2 or x depending on the field to be $GF(p)$ or $GF(2^n)$, respectively. These two operations obviously preserve our pair of congruences.

Hence, once we show that the WHILE loop ends with $D = 1$, it is clear that W then yields $XY^{-1} \bmod p$. However, the iterates of the values (C,D) are calculating the greatest common divisor of Y and p in the appropriate ring: $\mathbb{Z}$ or $(\,GF(2)\,)[x]$; that is, in any iteration, (C,D) is the value of consecutive remainders in a modified extended

Euclidean algorithm applied to (Y, p). Since this pair are relatively prime, the GCD calculation ends with one of C, D being a unit, the other being zero. On the other hand, it is easily seen that D is such that $d_0 = 1$ throughout any division phase. Hence, we eventually find $C = 0$ and $D = 1$. $\square$

2.3. Hardware Architecture of The Unified Modular Divider

Figure 4.1 shows the top level organization of the unified modular divider that implements the UMD algorithm. The main functional blocks are *Registers, Swapping Network (Multiplexers), Control and Datapath.*

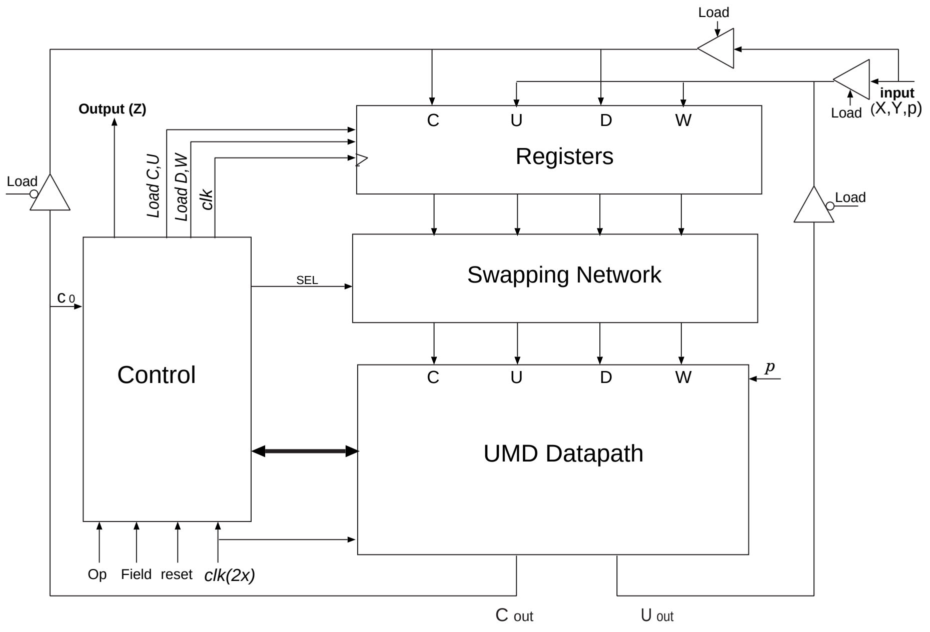


FIGURE 2.2: Top level organization of the modular divider which implements the UMD Algorithm.

The registers C, U, D, and W are initialized with the inputs (X,Y,p) when $Load =$ 1 through three-state buffers. When $Load = 0$, the registers receive their values from

U_{out} and C_{out} coming from the datapath. U_{out} is fed back to either U or W registers depending on *Load U* and *Load W*, respectively. Also, C_{out} is fed back to C or D registers depending on *Load C* or *Load D*, respectively. All these signals are generated by the control block.

The swap operations ($C \Leftrightarrow D$, $U \Leftrightarrow W$) are realized by the Swapping Network which is a set of two-input muxes, controlled by the SEL signal provided by the control block and takes its value based on the value of δ (kept internally). More detail are provided in the dtatpath Section.

2.3.1. Adders Scheduling for Efficiency

The UMD algorithm performs in the worst case (*else* part of the algorithm) 3 additions in each iteration which are shown in Figure 2.3. Using 3 adders will increase the area of the design significantly especially for large precision inputs. Another alternative is to use a single adder in more than one clock cycle to complete one iteration. Such a solution would increase the overall time to compute the division but would be a solution when the area is too restrictive. Therefore for this implementation of an isolated division unit the use of two adders is the best choice.

In this worst case scenario (Figure 2.3), there is data dependency between additions $A1$ ($U + k * W$) and $A2$ ($U + u_0 * p$). Therefore, one iteration is complete only after two consecutive additions are performed. If we assign addition $A1$ to one adder (adder1) and addition $A2$ to another adder (adder2), each adder will be working for only half of the clock cycle time. Based on this observation, we propose a solution that uses adder1 to compute addition 3 ($A3$) in the second half of the iteration cycle, while adder2 is computing $A2$. This solution requires a register or latch between the two adders, clocked at twice the clock frequency at which iterations are executed.

As can be seen from Figure 2.3, Adder1 receives the operands to compute $A1$ during phase 1 (ϕ_1). At the end of ϕ_1, the adder output is registered and another set of input values is applied to compute $A3$ during ϕ_2. Note that during ϕ_1 the Adder2 is

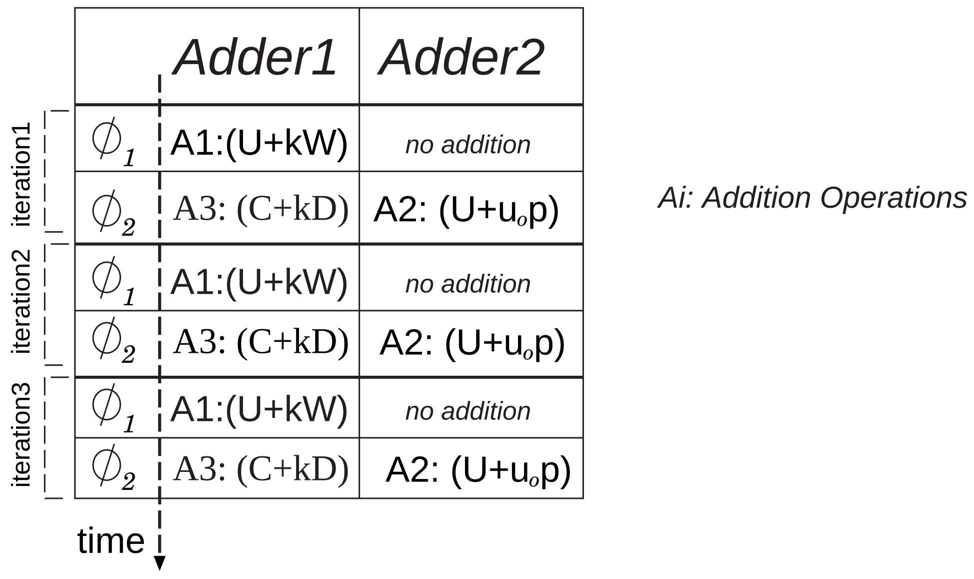


FIGURE 2.3: Scheduling of adders in the modular divider that implements the UMD algorithm

not used anyway because the signals are still propagating in the first half of the circuit. Another observation is that phase ϕ_2 can be shorter than ϕ_1 in order to keep the hardware units working most of the time.

2.3.2. Datapath

An n-bit datapath was designed to support the computations described by the UMD algorithm and it is shown in Figure 4.2. Each iteration of the algorithm is implemented in one clock cycle. The critical path delay (the clock cycle time) is determined by the datapath and control block, and it will be addressed in more detail in Section 2.3.3.

The proposed datapath uses two Carry-Save Unified Adders (CSUAs) to perform addition in both $GF(p)$ and $GF(2^n)$ fields. The CSUA is basically formed by dual-field adders which were described in [6] for a (3,2) design (3 inputs and 2 outputs) and in [43] for a (4,2) design. The (3,2) dual-field adder is similar in complexity to a full-adder and it performs bit addition with and without carry. This functionality is enabled by the

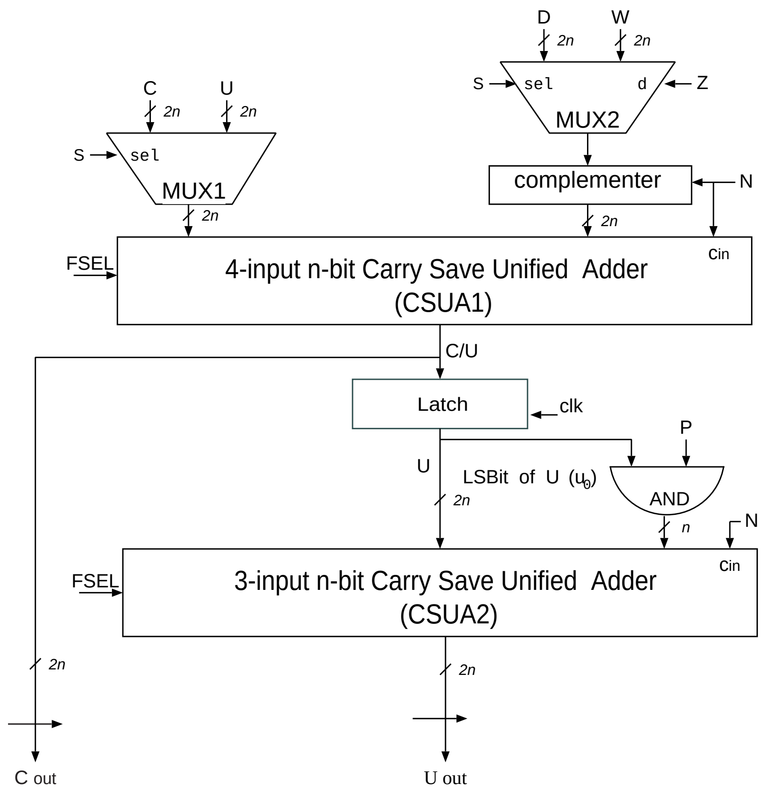


FIGURE 2.4: The Unified Datapath of the Modular Divider

input $FSEL$ (Field Select). When $FSEL = 0$, the carry out bits are forced to 0 and the dual-field adder performs bitwise modulo-2 addition of its inputs. When $FSEL = 1$, the dual-field adder performs the bitwise addition with carry (addition in $GF(p)$). Another implementations of unified adders can be used as the one proposed in [44].

CSUA1 was implemented using (4,2) dual-field adders and CSUA2 was implemented using (3,2) dual-field adders. The use of redundant form of the operands enables the circuit to have a critical path that is less sensitive to the operand precision. The addition time is less than the time for non-redundant adder, especially for large precision. A binary vector X is represented in CS form by two vectors XC and XS such that $X = XC + XS$. Therefore, the cost of CS representation comes from more registers and buses.

The three control lines: S, Z, and N in $MUX2$ corresponds to select, zero, and negate, respectively. When $Z = 1$ the output of the mux is forced to zero regardless of S. $N = 1$ produces a bit-complement of the input. Since we are dealing with numbers in two's complement represented in CS form, the change of sign is done by complementing each vector and adding 1. Thus, N is inserted as carry input into both CSUAs to get the change of sign operation in this system.

The latch between the two carry-save unified adders lets the information at its input pass through during ϕ_1 and holds the information at its output when it is ϕ_2.

The UMD algorithm computes the modular division in $GF(p)$ when $Field = GF(p)$. The select signal S is synchronized with the latch. $MUX2$ is used to implement $k * D$ and $k * W$, where $k \in$ {-1,1}. In the case $k = -1$, the negative D and the negative W are obtained by setting $N = 1$. Both signals Z and N are synchronized with the main clock (clk).

If C is even, then it is shifted right one bit and the counter δ is decremented by one. If not, we test δ, if it is negative, the circuits swap the values of C and D, and U and W, and change the sign of δ. The swap operation is performed by the Swapping Network that precedes the datapath and takes its inputs from the C, U, D, and W registers.

The test $(C + D)$ mod $4 \neq 0$ can be implemented using a small two-level gate network.

The addition $U := (U + k*W)$ which corresponds to $A1$ in Figure 2.3, is performed in the first phase of the clock signal (ϕ_1) using the CSUA1. During ϕ_2, two separate additions happen: $C := (C + k*D)$ ($A3$ in Figure 2.3) using CSUA1, and $U := (U + u_0 * p)$ ($A2$) using CSUA2. Both outputs are shifted to the right by one bit to complete the algorithm operations.

An AND gate is used to select between the value 0 or the modulus p depending whether U is even or odd, respectively.

If the algorithm is computing the modular division in $GF(2^n)$, the same procedure described above is followed, except that the test $(C + D)$ mod $4 \neq 0$ is not applicable

($Field = 0$). For both fields, the computation is done when $C = 0$, and the result is $Z = W$.

It can be shown that the UMD algorithm does not change the values of the operands once $C = 0$. Therefore, the test $C = 0$ can take several clock cycles. Another observation shows that the non-redundant representation of $C = 0$ takes only some particular values, what makes this test easier. So, using these two features we can make the test of zero for the CS representation simple and multi-cycle, allowing the design to be fast without a significant increase in area. Another possibility is to use counters to estimate when C reaches 0.

The Swapping Network shown in the datapath is composed of two-input muxes. The control signal (SEL) selects between the inputs. The two possible configurations of the Swapping Network are shown in Figure 2.5 (when SEL=0 or 1).

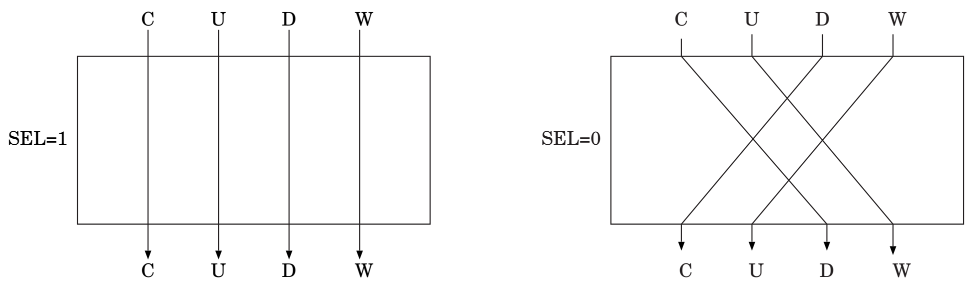


FIGURE 2.5: The two possible configurations of the Swapping Network.

2.3.3. *Improving The System Performance*

Figure 2.6 shows the critical path of the unified divider which will determine the clock period of the design.

$$clock\ period = max(delay\phi_1, delay\phi_2)$$

From the figure it is clear that ϕ_1 is longer than ϕ_2. There are two possibilities for the delays in ϕ_2 as shown in Figure 2.6, coming from the paths that include CSUA1 or CSUA2. Noticing that the delay of CSUA2 is smaller than the delay of CSUA1, the upper path is longer, so it is considered as the delay of ϕ_2.

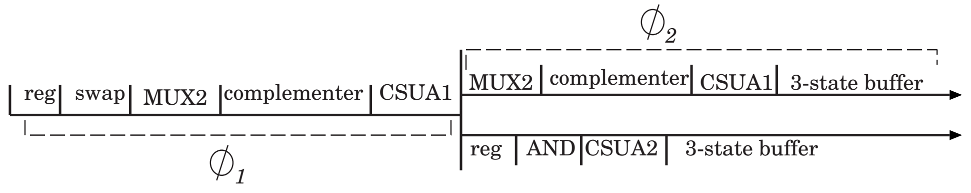


FIGURE 2.6: The delay paths of the Modular Divider.

Since that $delay\phi_1 > delay\phi_2$, the delay of ϕ_1 determines the clock period of $clk2x$, which is half the clock period of clk. In this case clk will have a 50% duty cycle. More performance could be extracted from the circuit if ϕ_2 could be made shorter. However, such a solution would involve critical implementation details for the design of the clock signal generator and clock distribution network.

3. NOVEL UNIFIED MODULAR DIVISION AND MULTIPLICATION ALGORITHM (UDMA) IN $GF(P)$ AND $GF(2^N)$.

This Chapter presents a novel Unified modular Division and Multiplication Algorithm (UDMA) in $GF(p)$ and $GF(2^n)$. To the best of our knowledge, the UDMA is the first algorithm to compute modular division and multiplication in both $GF(p)$ and $GF(2^n)$ fields (*Unified*). The UDMA is based on the Extended Binary GCD algorithm for modular division, and on the Montgomery's method for modular multiplication. In the next Section we present the UDMA, followed by details about the multiplication mode in Section 3.2. The modular division mode of the UDMA is explained in Section 3.3. A mathematical proof of the algorithm is presented in Section 3.4.

3.1. Unified Algorithm for Modular Division and Montgomery Modular Multiplication

Figure 3.1 shows the Unified modular Division/Multiplication Algorithm (UDMA) for both $GF(p)$ and $GF(2^n)$ fields. Up to our knowledge, this is the first algorithm that integrates the computation of modular division and Montgomery modular multiplication in both $GF(p)$ and $GF(2^n)$ fields. The UDMA mode of operation is controlled by the input Op (div or $mult$), and the finite field is controlled by the input $Field$ ($GF(p)$ or $GF(2^n)$). For simplicity, the polynomials $X(x)$, $Y(x)$, and $p(x)$ are denoted as X, Y, and p, respectively, which corresponds to the bit-vector representation of these polynomials.

Most of the arithmetic operations in the algorithm are common to both modes of operation. The initialization of variables depends on that division or multiplication being performed by the algorithm. For a given field, all the additions/subtractions are done in the field, besides the arithmetic operations on δ (decrement and change of sign) which are always integer operations. Decrementing δ in both division and multiplication modes, can be implemented using fast up/down counter as the ones proposed in [41].

Function: Modular Division and Multiplication in $GF(p)$ and $GF(2^n)$

Inputs: $0 \leq X < p$, $0 < Y < p$, $2^{n-1} < p < 2^n$, $Field$, Op, n

Output: $Z = XY2^{-n}\ mod\ p$ when $Op = mult$, $Z = \frac{X}{Y}\ mod\ p$ when $Op = div$.

Algorithm:

$C = Y$.

```
IF $Op = mult$ THEN                          /* $Multiplication\ Mode$ */
    $D = 0$, $U = 0$, $W = X$, $\delta = n$
ELSE                                         /* $Division\ Mode$ */
    $D = p$, $U = X$, $W = 0$, $\delta = 0$
END IF;
WHILE $[(C \neq 0$ AND $Op = div)$ OR $(\delta \neq 0$ AND $Op = mult)]$
    IF $c_0 = 0$ THEN
        $C := C >> 1$
        $\delta := \delta - 1$          /* $Integer\ Operation$ */
    ELSE
        $k = 1$
        IF $(Op = div)$ THEN
            IF $\delta < 0$ THEN $C \Leftrightarrow D$, $U \Leftrightarrow W$, $\delta := -\delta$   END IF; /* $Swapping$ */
            IF$((C + D)\ mod\ 4 \neq 0$ AND $Field = GF(p))$ THEN $k = -1$
            ELSE    $\delta := \delta - 1$    END IF;
        ELSE        /* $Op = mult$ */
            $\delta := \delta - 1$
        END IF;
        $C := (C + k * D) >> 1$,   $U := (U + k * W)$
    END IF;
    $U := (U + u_0 * p) >> 1$
END WHILE;
IF $Op = div$ THEN $Z := W$    ELSE $Z := U$
END IF;
```

FIGURE 3.1: Unified modular Division/Multiplication Algorithm (UDMA) for $GF(p)$ and $GF(2^n)$.

In this work, we used a similar counter to the one used in [29] as explained in the next Chapter. The $>>$ operator indicates a 1-bit right shift operation.

The unified division/multiplication algorithm (UDMA) presented in this section, computes modular multiplication using Montgomery's method. Section 3.2 shows the operation of the UDMA when performing Montgomery multiplication. The correctness of the UDMA is proven mathematically in Section 3.4..

The UDMA computes modular division using the same structure used by the modular division algorithm presented in the last Chapter, which in turn, is based on the Extended Binary GCD Algorithm [40, 27]. The operation of the UDMA during computing modular division is shown in Section 3.3.

More details about the operation and hardware implementation of the UDMA are presented in the next Chapter.

3.2. Multiplication Mode

To perform Montgomery multiplication using the UDMA, we set the variable $Op = mult$, and we choose the field to be either $GF(p)$ or $GF(2^n)$. Figure 3.2 shows the resulting operations performed by the UDMA during computing Montgomery multiplication after removing the control signals and tests that are required during division mode.

The UDMA performs n iterations to compute Montgomery multiplication using an n bit modulus p. The counter δ is initialized with value n, and in each iteration it is decremented by one. The variables used in the algorithm are initialized as: $C = Y$, $D = 0$, $U = 0$, and $W = X$. The partial product U is reduced *mod p* in each iteration. In both fields, addition is used in the operations that update C ($k = 1$ always).

Table 3.1 shows the operation of the unified modular division/mulitipliaction algorithm when performing Montgomery multiplication in either $GF(p)$ or $GF(2^n)$ fields. Shifting the multiplier C happens in each iteration (notice that the value of D stays

$c_0 = 0$ (**THEN**)	$c_0 = 1$ (**ELSE**)	
$U := (U + u_0 * p) \;>>\; 1$	$U := (U + k * W)$	$U := (U + u_0 * p) \;>>\; 1$
$C \;>>\; 1$, $\delta = \delta - 1$		

TABLE 3.1: The operations performed by the UDMA during Montgomery multiplication

zero during computing Montgomery multiplication, and so the expression $C := (C + k * D) \;>>\; 1$ is reduced to $C \;>>\; 1$).

Notice that the addition in $GF(2^n)$ is done without carry propagation. The final result is ready ($Z = U$) when $\delta = 0$. If $Z > p$ then $Z = U - p$. In order to use the intermediate result of one multiplication as the input to another one without subtracting p, we need to use two extra bits of precision ($n + 2$ bits) [38, 33].

3.2.1. Numerical Example:Computing Montgomery Multiplication Using The UDMA

To explain the operation of the UDMA more clearly, we show in Table 3.2 a numerical example to compute Montgomery multiplication using the UDMA. The computations are done in the prime field with $p = 503$, (and so the precision $n = 9 \Rightarrow r = 2^n$). We want to compute $(483 \times 301 \times r^{-1}$ *mod* p. The Table shows the intermediate values of the variables: $delta, C, U$, and W. Notice that D is set to zero during the multiplication mode. The operations performed by the UDMA is also shown ($C \;>>\; 1$ or $U := (U + k * W)$). The reduction *mod* p is performed in each iteration.

The computation is completed when the counter $\delta = 0$. The result can be read from U,$301 \times 483 \times 2^{-9}$ *mod* $503 = 393$. Notice that C is also zero at the end. This indicates that the algorithm scanned all the n bits of the multiplier.

Operations	δ	C	U	W
Initialization	9	301	0	483
$ELSE$	8	150	493	483
IF	7	75	498	483
$ELSE$	6	37	742	483
$ELSE$	5	18	864	483
IF	4	9	432	483
$ELSE$	3	4	709	483
IF	2	2	606	483
IF	1	1	303	483
$ELSE$	**0**	0	**393**	483

TABLE 3.2: Montgomery multiplication example by the UDMA in $GF(p)$

3.3. Division Mode

The unified modular division/multiplication algorithm computes modular division when the variable $Op = div$ in both fields depending on the value of the variable $field$. We show in Figure 3.3 the algorithm during modular division computations.

The variables are initialized as: $C = Y$, $D = p$, $U = X$, $W = 0$, and $\delta = 0$. If the division is computed in $GF(p)$, UDMA tests the least significant two bits of C and D ($(C + D)\ mod\ 4 \neq 0$) to conditionally subtracts C from D (set $k = -1$). Otherwise, C is always added to D in both fields. The division is completed when $C = 0$, and the final result is available in W. Table 3.3 summarizes the operations performed by the UDMA when computing modular division.

3.3.1. Numerical Example:Computing Modular Division Using The UDMA

Table 3.4 shows a modular division example by the UDMA. The computations are done in $GF(2^4)$ (we choose a small field for simplicity) with an irreducible polynomial $p(t) = t^4 + t^3 + 1$. The algorithm computes $(\frac{t^3+t+1}{t^3+1}\ mod\ p(t))$. The intermediate values

Function: Multiplication Mode of The UDMA

Inputs: $0 \leq X < p$, $0 < Y < p$, $2^{n-1} < p < 2^n$, $Field$, n

Output: $Z = XY2^{-n} \ mod \ p$

Algorithm:

$C = Y$.

```
 $Op = mult$ THEN
     $D = 0$, $U = 0$, $W = X$, $\delta = n$
 WHILE ( $\delta \neq 0$ )
     IF $c_0 = 0$ THEN
         $C := C \ >> \ 1$
         $\delta := \delta - 1$            /* $Integer\ Operation$ */
     ELSE
         $k = 1$
         $\delta := \delta - 1$
         $C := (C + k * D) \ >> \ 1$,   $U := (U + k * W)$
     END IF;
     $U := (U + u_0 * p) \ >> \ 1$
 END WHILE;
 $Z := U$
```

FIGURE 3.2: Computing Montgomery multiplication in $GF(p)$ and $GF(2^n)$ using the UDMA.

$c_0 = 0$ (**THEN**)	$c_0 = 1$ (**ELSE**)
$C \ >> \ 1$ $U := (U + u_0 * p) \ >> \ 1$	$C := (C + k * D) \ >> \ 1$ $U := (U + k * W)$ $U := (U + u_0 * p) \ >> \ 1$

TABLE 3.3: The operations performed by the UDMA during modular division

Operations	δ	C	D	U	W
Initialization	0	t^3+1	t^4+t^3+1	t^3+t+1	0
$ELSE$	-1	t^3	t^4+t^3+1	t^3+1	0
$THEN$	-2	t^2	t^4+t^3+1	t^3	0
$THEN$	-3	t	t^4+t^3+1	t^2	0
$THEN$	-4	1	t^4+t^3+1	t	0
$ELSE$	3	t^3+t^2	1	1	t
$THEN$	2	t^2+t	1	t^3+t^2	t
$THEN$	1	$t+1$	1	t^2+t	t
$ELSE$	0	1	1	t	t
$ELSE$	-1	**0**	1	0	**t**

TABLE 3.4: A modular division by the UDMA in $GF(2^4)$

of the variables: $delta, C, D, U$, and W are shown in the Table with the corresponding performed operation by the UDMA. The Table also shows which portion of the algorithm was carried out ($THEN$ or $ELSE$). Notice that the modular reduction step ($U := (U + u_0 * p) >> 1$) is performed every iteration.

As can be seen form the Table, the computations are done when $C = 0$, and the final result is $W = t$.

3.4. Mathematical Proof During Computing Montgomery Multiplication

The UDMA performs modular division based on the Extended Binary GCD algorithm and the unified modular division algorithm which we proposed with its mathematical proof in Chapter 2, and it will not be mentioned here to avoid repetition. In this Section we prove the correctness of computing Montgomery multiplication by the UDMA.

Since the UDMA operates in both fields, we use ξ to denote either 2, or x, depending upon the context of field being $GF(p)$ or $GF(2^n)$, respectively.

Input X, Y as well as n and p, we are to output $Z = XY\xi^{-n}$.

3.4.1. Product modulo p

The multiplication algorithm employs a standard idea. First, writing Y as $\sum_{j=0}^{n-1} y_j \xi^j$, we have

$$\begin{aligned}
XY\xi^{-n} &= X\xi^{-n} \sum_{j=0}^{n-1} y_j \xi^j \\
&= \sum_{j=0}^{n-1} y_j X\, \xi^{j-n} \\
&= (\cdots(((y_0 X\xi^{-1} + y_1 X)\xi^{-1} + y_2 X)\xi^{-1} + \cdots + y_{n-2}X)\xi^{-1} + y_{n-1}X \\
&\equiv (\cdots(((y_0 X + v_0 p)\xi^{-1} + y_1 X + v_1 p)\xi^{-1} + y_2 X + v_2 p)\xi^{-1} + \cdots \\
&\qquad \cdots + y_{n-2}X + v_{n-2}p)\xi^{-1} + y_{n-1}X + v_{n-1}p \mod p \ ,
\end{aligned}$$

where the $v_i \in \{0,1\}$. In this article, single shifts to the right correspond to multiplication by ξ^{-1}. In the algorithm, U begins at zero; the value of Y is given to C, and that of X to W. Thus, whenever $y_j = 0$, we find that U is replaced by $(U + u_0 * p)\xi^{-1}$. Whenever $y_j = 1$, U is replaced by $[(U + X) + (u_0 + x_0) * p]\xi^{-1}$. We conclude that U evaluates the product $XY\xi^{-n} \bmod p$ in the form displayed above.

3.4.2. Product is Reduced

All field elements are represented by the standard complete set of residue class representatives, $\{0, 1, \ldots, p-1\}$, or $\{0, 1, x, x+1, \ldots\}$ except, of course, p or $p(x)$. Rather than evaluate the product in either ring $\mathbb{Z}$ or $(\,GF(2)\,)[x]$ and then in a final step reduce modulo the appropriate prime element p, the algorithm uses the standard "interleaving" process for reduction. Indeed, we claim that at the end of each iteration (of the WHILE loop), U is reduced.

Initially, $U = 0$. For a proof by induction, we thus assume that U is reduced at the beginning of any arbitrary iteration. Now, upon completion of the iteration, (1) U is replaced by (1) $(U + u_0 p)\xi^{-1}$; or, (2) we first find $U + X$ (in reduced form) and then

$[(U+X)+(u_0+x_0)*p]\xi^{-1}$, where the multiplication by ξ^{-1} is effected by a right shift. Since for any reduced element say V, $(V+v_0p)\xi^{-1}$ is easily seen to be reduced, we are done.

Function: Division Mode of The UDMA

Inputs: $0 \leq X < p$, $0 < Y < p$, $2^{n-1} < p < 2^n$, $Field$

Output: $\frac{X}{Y}$ $mod\ p$

Algorithm:

```
$C = Y$.
 $D = p$, $U = X$, $W = 0$, $\delta = 0$
 WHILE ($C \neq 0$)
     IF $c_0 = 0$ THEN
         $C := C \ >> \ 1$
         $\delta := \delta - 1$            /* Integer Operation */
     ELSE
         $k = 1$
             IF $\delta < 0$ THEN $C \Leftrightarrow D$, $U \Leftrightarrow W$, $\delta := -\delta$   END IF; /* Swapping */
             IF(($C + D$) $mod$ $4 \neq 0$ AND $Field = GF(p)$) THEN $k = -1$
             ELSE    $\delta := \delta - 1$    END IF;
         $C := (C + k * D) \ >> \ 1$,   $U := (U + k * W)$
     END IF;
     $U := (U + u_0 * p) \ >> \ 1$
 END WHILE;
 $Z := W$
```

FIGURE 3.3: Computing modular division in $GF(p)$ and $GF(2^n)$ using the UDMA.

4. SCALABLE AND UNIFIED MODULAR DIVIDER/MULTIPLIER HARDWARE DESIGN.

In this Chapter, we propose a Unified Modular Divider/Multiplier hardware that implements the Unified modular Division and Multiplication Algorithm (UDMA) presented in Chapter 3. In the next Section, we present the overall hardware system architecture and the implementation details. The datapath of the proposed architecture has a variable word size that can be adjusted to meet the area and performance requirements. In Section 4.2, we modify the proposed divider/mulitper architecture to get a scalable hardware that can handle operands of any size.

4.1. Overall Hardware System Architecture

Figure 4.1 shows the top level architecture of the Unified Modular Divider/Multiplier (let us call it UMDM) that implements the UDMA. The main functional blocks are *Register File*, *Datapath*, and *Control*.

The following subsections provide the implementation details of the functional blocks of the UMDM and explains the design approaches used to implement the proposed architecture [45].

4.1.1. Register File

The register file has five registers ($R1$ to $R5$). Since the computations are done in Carry-Save form [46], each intermediate variable (C, U, D, W) is represented in two vectors (sum,carry). So, the registers inside the register file are designed to store two n-bit vectors. In other words, the i^{th} register R_i is represented as $R_i = (sum, carry) = (R_is, R_ic)$.

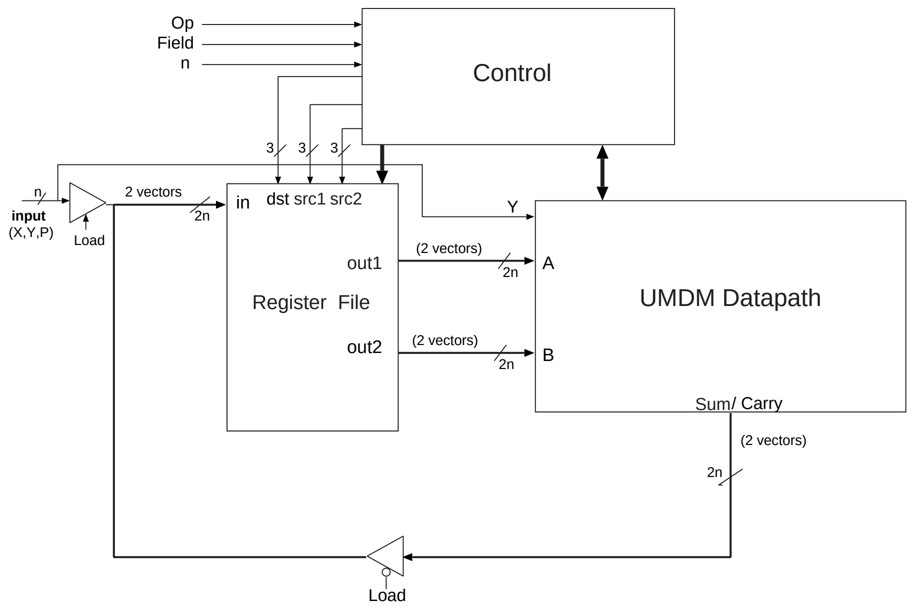


FIGURE 4.1: Top Level Hardware Architecture of the Unified Modular Divider/Multiplier (UMDM).

The register file has one input, and two output ports. The Control block provides the register file with the signals necessary to perform reading/writing operations. The 3-bit signal *dst* determines the destination register to be written. The signals *src1, src2* (3-bits each), specify the registers to be read at output ports *out1, out2*, respectively.

4.1.2. UMDM Datapath

The n-bit datapath implementing the UDMA is shown in Figure 4.2. Each “while iteration” of the algorithm is implemented in one clock cycle for multiplication mode, and in three clock cycles for division mode if C is odd, and 2 clock cycles if C is even, as explained later.

The proposed datapath has two inputs (A,B) represented in carry-save form as A=(As,Ac) and B=(Bs,Bc) which receive their values from the output ports ($out1, out2$) of the register file, respectively.

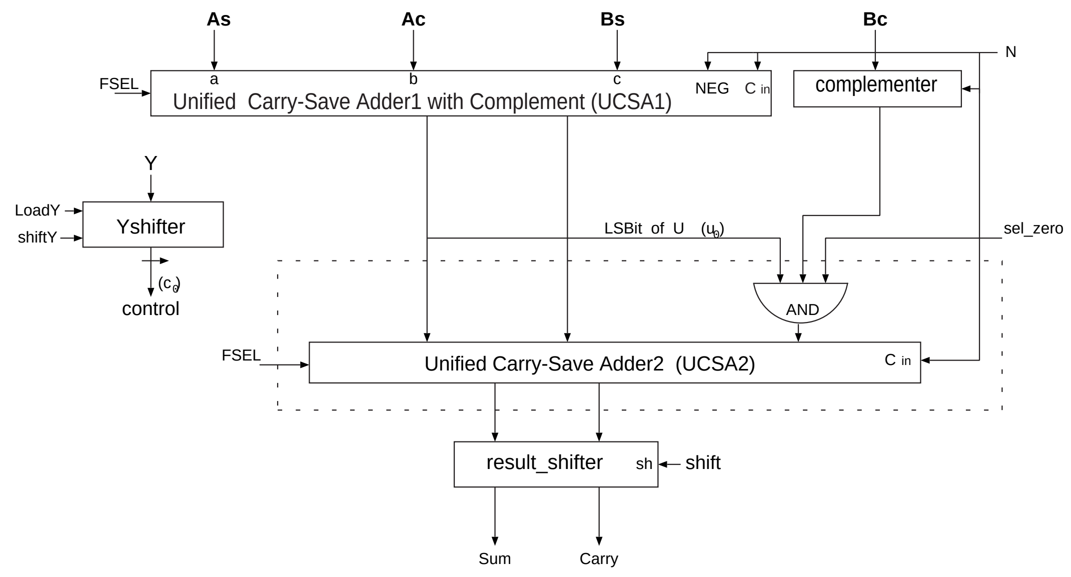


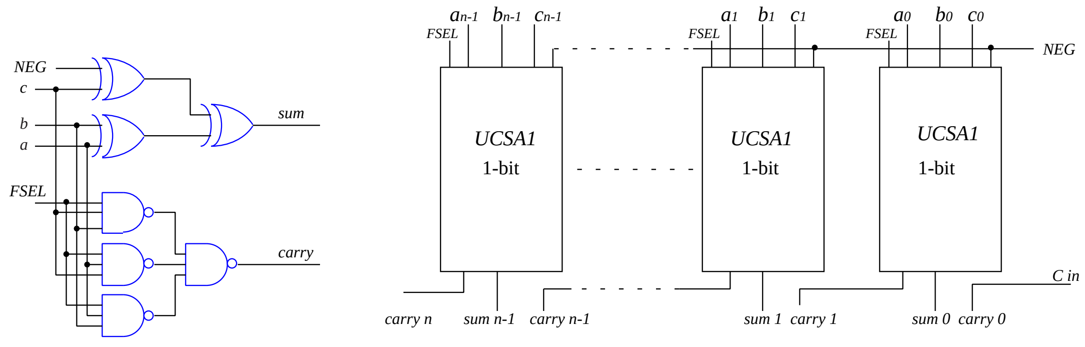
FIGURE 4.2: Unified datapath of the Modular Divider/Multiplier (UMDM datapath).

The main components of the datapath are two (3-2) Unified Carry-Save Adders (UCSAs) used to perform addition in either $GF(p)$ or $GF(2^n)$ fields. The Unified Adders (UAs) [6, 47] are similar in complexity to full-adders and are capable of doing bit addition with or without carry. It has an input called *FSEL* (Field Select) that enables this functionality.

The unified adder may be used to implement a redundant or non-redundant adder which may also be a full-precision or multi-precision adder. The use of non-redundant form of the operands and results reduces the register area but increases significantly the addition time (because of carry propagation), when compared with redundant adders. We decided to use Carry Save adders to make the addition time constant and independent of the operand's precision.

4.1.2.1. Unified Carry-Save Adders

The first adder in the datapath is a Unified Carry-Save Adder with complement (UCSA1). Figure 4.a shows the bit slice diagram for this adder and Figure 4.b shows the connection of n slices to form an n-bit adder.



(a) *bit slice* **(b)** *n-bit adder*

FIGURE 4.3: Unified Carry-Save Adder with Complement (UCSA1) for 1-bit and n-bit precision.

Equation 4.1.2.1. defines the outputs of the UCSA1 in terms of the inputs. The input signal *FSEL* determines in which field the addition or subtraction of operands are performed. Note that in $GF(2^n)$ addition is the same as subtraction.

$$(sum, carry) = \begin{cases} a{+}b{+}c & \text{, when } NEG = 0 \text{ and } Cin = 0 \\ a{+}b\text{-}c & \text{, when } NEG = 1 \text{ and } Cin = 1 \end{cases}$$

As we can see from Figure 4.3, the change of sign operation inside the UCSA1 is done in parallel with the addition operation, and so, it does not add to the critical path.

The *AND* gate shown at the top of UCSA2 in Figure 4.2 is used to select between the value 0 or the modulus p depending whether U is even or odd (this is indicated by testing u_0), respectively. The input signal *sel_zero* when asserted forces the third input of the UCSA2 to zero.

The delay of the UMDM datapath ($t_{datapath}$) is determined by the delay of the two unified adders, the delay of the AND gate between them, and the delay of the $result\_shifter$ which has the delay of 2-input multiplexer ($t_{MUX} \simeq t_{XOR}$). Subsection 4.1.2.2. describes the shifters used in the datapath. By integrating the AND gate with the second adder (shown in dashed box in Figure 4.2), its delay will not add to the path delay as shown in Figure 4.4. Figure 4.4 shows the integration of the AND gate to the UCSA2 at the bit level and as an n-bit adder. Knowing that each UCSA has a delay of a full adder ($t_{FA} = 2t_{XOR}$), we get:

$$t_{datapath} = t_{USCA1} + t_{UCSA2} + t_{result\_shifter} = 4t_{XOR} + t_{MUX} = 5t_{XOR}$$

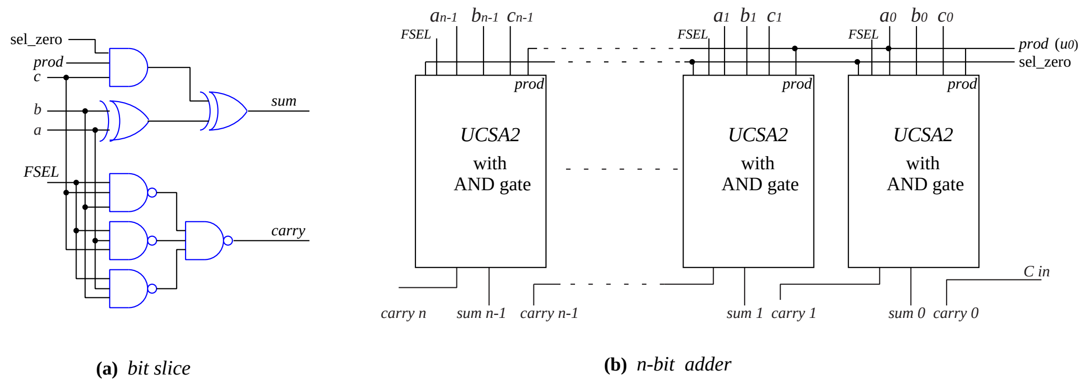


FIGURE 4.4: Unified Carry-Save Adder with integrated AND gate for 1-bit and n-bit precision

4.1.2.2. Shifters

The *Yshifter* shown in Figure 4.2 is a shift register that is used to implement the 1-bit right shift operation ($C \;\; >> \;\; 1$) only in the multiplication mode. The *Yshifter* is loaded with the input Y (multiplier) when $laodY = 1$. In both cases when C (remember that $C = Y$) is even or odd, it is shifted right by 1-bit when $shiftY = 1$. The least significant bit of the shifted C goes to the control section to perform the test $c_0 = 0$.

The outputs of the datapath (*Sum* and Carry) are shifted 1-bit to the right by correct wiring. The $result\_shifter$ at the output of the UCSA2 is used to choose between these outputs and their shifted values. This module is implemented as two 2-input multiplexers with select line $shift$. When $shift = 1$, the shifted outputs are selected.

4.1.3. Control Block

This subsection describes the Control block and the system operation during division and multiplication modes and the design techniques used to implement the proposed UMDM architecture.

When *Load* = 1, the registers inside the register file are accessible for external input (from the user) and are initialized with the inputs X,Y, and p depending on the operation to be performed by the algorithm.

While the algorithm is in division mode, the test $C \neq 0$ must be done. The vector C is in Carry Save form. Simulation results show that $C = 4$ or -4 at the end of computation. If more iterations than required are performed, the result is still correct. In the proposed UMDM design, the test $C \neq 0$ is replaced by testing the bit vector C for specific values (4 or -4), and no need to compare all the bits of C with zero.

4.1.3.1. Multiplication Mode

The proposed multiplier/divider performs one iteration of the algorithm in each clock cycle when computing Montgomery multiplication in both fields. Only three registers are used. Table 4.1 shows the loading phase for multiplication and division (initialization of the intermediate variables) as described by the algorithm.

Multiplication		*Division*	
Variables/parameters represented by R	Loading the RF $R \leftarrow (Rs, Rc)$	Variables/parameters represented by R	Loading the RF $R \leftarrow (Rs, Rc)$
W and p	$R_1 \leftarrow (X, p)$	W	$R_1 \leftarrow (X, 0)$
U	$R_2 \leftarrow (0, 0)$	U	$R_2 \leftarrow (0, 0)$
p	$R_3 \leftarrow (0, p)$	D	$R_3 \leftarrow (p, 0)$
Variable	Loading the Shifter	C	$R_4 \leftarrow (Y, 0)$
C	YShifter $\leftarrow (Y)$	p	$R_5 \leftarrow (0, p)$

TABLE 4.1: Loading phase for multiplication and division

Table 4.2 shows the operation of the unified divider/mulitplier when performing Montgomery multiplication in either $GF(p)$ or $GF(2^n)$ fields. It shows the main control signals to the register file and the datapath to compute multiplication. I assumed that the signals which are not mentioned in the Table have their values equal to zero. Depending on C being even or odd, a different set of signals are used. The Table also shows the interpretation of these signals on the datapath, and the corresponding operation in the UDMA. When $c_0 = 1$, two additions ($U := (U + k * W)$ and $U := (U + u_0 * p) \; >> \; 1$) are performed by the datapath.

As mentioned in subsection 4.1.2.2., the variable C is shifted right one bit using the $Yshifter$ in every cycle ($shiftY \leftarrow 1$). The counter δ is initialized with n to indicate the number of iterations. Since δ is decremented by one in each iteration ($dec\_delta = 1$), the computation ends when $\delta = 0$, and the result can be read form R_2.

4.1.3.2. Division Mode

Each modular division iteration in the proposed UMDM architecture requires 2 clock cycles if C is even and 3 clock cycles if C is odd independent of the field. The initialization of variables was shown in Table 4.1.

Let us assume that the algorithm is computing modular division in $GF(p)$. Tables 4.3 and 4.4 show the control signals during division when C is even and odd, re-

$c_0 = 0$ (**then**)		$c_0 = 1$ (**else**)	
Control Signals	Datapath Interpretation	Control Signals	Datapath Interpretation
$src1 \leftarrow (010)$	$(As, Ac) \leftarrow (Us, Uc)$	$src1 \leftarrow (010)$	$(As, Ac) \leftarrow (Us, Uc)$
$src2 \leftarrow (011)$	$(Bs, Bc) \leftarrow (0, p)$	$src2 \leftarrow (001)$	$(Bs, Bc) \leftarrow (X, p)$
$dst \leftarrow (010)$	store the result in U	$dst \leftarrow (010)$	store the result in U
$shiftY \leftarrow 1$	$C >> 1$	$shiftY \leftarrow 1$	$C >> 1$
$shift \leftarrow 1$	the result $>> 1$	$shift \leftarrow 1$	the result $>> 1$
$dec\_delta \leftarrow 1$	$\delta = \delta - 1$	$dec\_delta \leftarrow 1$	$\delta = \delta - 1$
Computes:	$U := (U + u_0 * p) >> 1$	$U := (U + k * W)$, $U := (U + u_0 * p) >> 1$	

TABLE 4.2: The operation of the UMDM during Montgomery multiplication

spectively. Table 4.3 shows when δ is positive, the expression $U := (U + u_0 * p) >> 1$ is computed in the first clock cycle. In the second cycle, C is shifted right by 1-bit ($C >> 1$). The reason why $C >> 1$ happened at cycle 2, is to enable the test for $c_0 = 0$ which decides on the control signals of the next iteration.

The Control during division has two major "states": *Original* (not swapped) and *Swapped*. The control goes back and forth between these states only when $c_0 = 1$ and $\delta < 0$. The sign of δ is made positive ($\delta = -\delta$) when this condition happens (realized by the control signal *negate_delta* as shown in Table 4.4). When the control state is *Original*, the variables W, U, D, C are read from the registers R_1, R_2, R_3, R_4, respectively. Also, the updated values of these variables are written to the above registers in the same order. On the other hand, when the control state is *Swapped*, the sources for the variables W and U are reversed (W now is being read from R_2 and U from R_1). The same thing applies for C and D, so C is read from R_3 and D from R_4. The same approach is used for writing to these variables. This way, the swap operation is realized by reading from/writing to the correct register. The next subsection explains an efficient hardware implementation of the counter δ.

The value of $k \in$ {-1,1} is determined by the control section depending on the result of the test $((C + D)\ mod\ 4 \neq 0\ AND\ Field = GF(p))$, which is denoted by

$c_0 = 0$ (**then**)				
	State1: Original		State2: Swapped	
Clock Cycle	Control Signals	Datapath Interpretation	Control Signals	Datapath Interpretation
cycle 1	$src1 \leftarrow (001)$	$(As, Ac) \leftarrow (Us, Uc)$	$src1 \leftarrow (010)$	$(As, Ac) \leftarrow (Ws, Wc)$
	$src2 \leftarrow (101)$	$(Bs, Bc) \leftarrow (0, p)$	$src2 \leftarrow (101)$	$(Bs, Bc) \leftarrow (0, p)$
	$dst \leftarrow (001)$	store result in $R_1(U)$	$dst \leftarrow (010)$	store result in $R_2(W)$
	$shift \leftarrow 1$	result $>>$ 1	$shift \leftarrow 1$	result $>>$ 1
	$U := (U + u_0 * p)$ $>>$ 1 is computed			
cycle 2	$src1 \leftarrow (100)$	$(As, Ac) \leftarrow (Cs, Cc)$	$src1 \leftarrow (011)$	$(As, Ac) \leftarrow (Ds, Dc)$
	$sel\_zero \leftarrow 1$	$(Bs, Bc) \leftarrow (0, 0)$	$sel\_zero \leftarrow 1$	$(Bs, Bc) \leftarrow (0, 0)$
	$dst \leftarrow (100)$	store result in $R_4(C)$	$dst \leftarrow (011)$	store result in $R_3(D)$
	$shift \leftarrow 1$	result $>>$ 1	$shift \leftarrow 1$	result $>>$ 1
	$dec\_delta \leftarrow 1$	$\delta = \delta - 1$	$dec\_delta \leftarrow 1$	$\delta = \delta - 1$
	$C := (C)$ $>>$ 1 is computed			

TABLE 4.3: The UMDM operation during computing division when $c_0 = 0$

($TEST$) in Table 4.4. In the case $k = -1$, negative D and negative W are obtained by setting $N = 1$. Since the $TEST$ is done on the least two significant bits of C and D, it can be implemented using a two-level gate network.

If the algorithm is computing the modular division in $GF(2^n)$, the same procedure described above is followed, except that the $TEST$ is not applicable ($Field = GF(2^n)$). For both fields, the computation is done when $C = 0$, and the result is $Z = W$.

4.1.3.3. Implementing δ Using Fast Counters

In both division and multiplication modes, the variable δ can be represented using a very fast up/down counter as the ones proposed in [41]. The counter used in this work is similar to the one in [29].

An up/down counter for hundreds of bits may have long carry/borrow propagation chains which implies in large critical path delay. To avoid that, the counter for δ uses a binary number H and a flag f, $f \in \{0, 1\}$. H is an n-bit vector and has the value

$c_0 = 1$ (**else**)				
	State1: Original		State2: Swapped	
Clock Cycle	Control Signals	Datapath Interpretation	Control Signals	Datapath Interpretation
cycle 1	$src1 \leftarrow (001)$	$(As, Ac) \leftarrow (Us, Uc)$	$src1 \leftarrow (010)$	$(As, Ac) \leftarrow (Ws, Wc)$
	$src2 \leftarrow (010)$	$(Bs, Bc) \leftarrow (Ws, Wc)$	$src2 \leftarrow (001)$	$(Bs, Bc) \leftarrow (Us, Uc)$
	$dst \leftarrow (001)$	store result in $R_1(U)$	$dst \leftarrow (010)$	store result in $R_2(W)$
	$shift \leftarrow 0$	result is not shifted	$shift \leftarrow 0$	result is not shifted
	$dec\_delta \leftarrow *$, *=1 when $TEST = 0$, otherwise *=0. $(\delta = \delta - 1)$			
	$U := (U + k * W)$ is computed			
cycle 2	$src1 \leftarrow (001)$	$(As, Ac) \leftarrow (Us, Uc)$	$src1 \leftarrow (010)$	$(As, Ac) \leftarrow (Ws, Wc)$
	$src2 \leftarrow (101)$	$(Bs, Bc) \leftarrow (0, p)$	$src2 \leftarrow (101)$	$(Bs, Bc) \leftarrow (0, p)$
	$dst \leftarrow (001)$	store result in $R_1(U)$	$dst \leftarrow (010)$	store result in $R_2(W)$
	$shift \leftarrow 1$	result $>>$ 1	$shift \leftarrow 1$	result $>>$ 1
	$negate\_delta \leftarrow *$, *=1 when $\delta < 0$, otherwise *=0. $(\delta = -\delta)$			
	$U := (U + u_0 * p) >> 1$ is computed			
cycle 3	$src1 \leftarrow (100)$	$(As, Ac) \leftarrow (Cs, Cc)$	$src1 \leftarrow (011)$	$(As, Ac) \leftarrow (Ds, Dc)$
	$src2 \leftarrow (011)$	$(Bs, Bc) \leftarrow (Ds, Dc)$	$src2 \leftarrow (100)$	$(Bs, Bc) \leftarrow (Cs, Cc)$
	$dst \leftarrow (100)$	store result in $R_4(C)$	$dst \leftarrow (011)$	store result in $R_3(D)$
	$shift \leftarrow 1$	result $>>$ 1	$shift \leftarrow 1$	result $>>$ 1
	$C := (C + k * D) >> 1$ is computed			

TABLE 4.4: The UMDM operation during division when $c_0 = 1$

$H = 2^{(-1)^f \cdot \delta}$. Note that H has a one-hot encoding of the value $|\delta|$. By using this approach, we decrement δ by a one-bit shift of H. H is shifted by 1-bit to the right when $f = 0$ (δ is positive), and by 1-bit to the left when $f = 1$ (δ is negative).

4.2. Scalable Divider/Multiplier Architecture

An arithmetic unit is called scalable if it can be reused in order to generate long-precision results independently of the datapath precision for which the unit was originally designed [6, 48, 43]. To speed up the arithmetic operations such as multiplication and division, various dedicated arithmetic modules (e.g., dividers, multipliers) were developed [14, 17, 23, 49], which use fixed-precision operands. They are fixed-precision designs because a module designed for n bits cannot be immediately used in a system which requires $k > n$ bits, forcing a complete redesign [36, 33]. As an example of a scalable unit is the multiplier presented in [6]. It used processing elements that can be adjusted in size and number in order to fit into a given area.

The scalability feature of the proposed divider/multiplier allows the datapath to handle operands of any size. Also, the variable word size of the datapth can be adjusted to meet the area and performance requirements.

Let the actual operand size be n, and let w be the word size of the datapath. We define the number of words in the operand as $e = \frac{n}{w}$. To carry out one iteration of the algorithm, all the e words will pass through the datapath. There is a data dependency between the e words, and some bits are needed at different times. So, these bits have to be stored and used at the appropriate time. More implementation details about the operation of the scalable design in both division and multiplication modes are presented below.

4.2.1. Implementation Details

To make the explanation more clear and easier to understand, let us take an example: let the operand size $n = 32$ bits, and the datapath word size, $w = 8$ bits. Then $e = \frac{32}{8} = 4$ words. All of these words will pass through the datapath in each iteration of the algorithm. Figure 4.5 shows the 4 words and the data dependency between them.

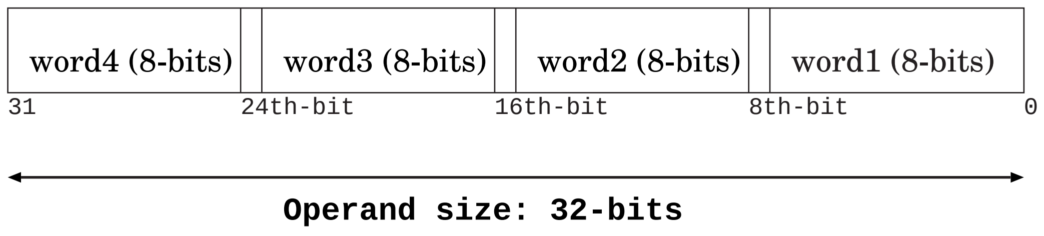


FIGURE 4.5: Data dependency between the words of the scalable design

The first word will be applied to the datapath in the first iteration. Depending on C being even or odd (testing c_0), it takes 2 or 3 cycles, respectively, to finish one division algorithm iteration. In order to shift the first word to the right, the 8^{th} bit of the operand (or bit 0 of the second word) is needed. This bit is stored in Flip-Flop to be used at the appropriate time. Once the computations are done on the first word, it is stored in the memory waiting for the rest of the words to be computed.

The same operations performed on the first word will be performed on the second, third and fourth words to complete the first algorithm iteration. Again to shift the second word to the right, the 16^{th} bit (or bit 0 of the third word) is required, and so, it is stored in a Flip-Flop. The same thing is done for the third word. In general, for the word number y, we need to store the $w * y$ bit. For example for the third word ($y = 3$), we store the ($8 * 3 = 24^{th}$) bit. To shift the last word to the right, the most significant bit is generated by a simple logic which depends on the carry save representation properties.

Since the variables D, U, W might be shifted during the computations, the appropriate bits of these vectors are also stored in Flip-Flops.

To carry out the next algorithm iteration, we again test the least significant bit of the first word of C, and apply the operations to all words, as done in iteration one.

The test ($C + D \ mod \ 4$), is performed in $GF(p)$ only, and it needs the two least significant bits of C and D. These four bits are stored in a Flip-Flops to be tested in the next iteration.

The procedure to compute Montgomery multiplication using the scalable design is almost the same to the above procedure described for computing modular division. Notice that it takes only one clock cycles to perform one Montgomery multiplication iteration as explained in Section 4.1.2.

5. EXPERIMENTAL RESULTS AND COMPARISONS.

This Chapter contains two categories of experimental results : (a) number of iterations and additions obtained from a *Maple* model, and it is presented in Section 5.1,and (b) the critical path delay results obtained by synthesis of the VHDL description of the algorithm presented in Section 5.2. The scalable design was synthesized using FPGA and the results are shown in Section 5.3. The generated results were compared with other known designs in this field.

5.1. The Number of Iterations

Maple was used to describe the proposed algorithm ($Alg1 = UDMA$) and the unified Montgomery inverse algorithm presented in [11] ($Alg2$). At least 100 *random samples* were used to verify each algorithm operation and obtain statistics.

The other division (inversion) algorithm being compared is the unified Montgomery inverse algorithm presented in [11] ($Alg2$). For consistency, no multiple-word calculation is considered here. For an $n-$bit input Y, $Alg2$ computes $Z = Y^{-1}2^k$, where $n \leq k \leq 2n$ is the number of algorithm iterations. A correction step is needed to get the inverse in the Montgomery domain ($Y^{-1}2^n$) or in the integer domain (Y^{-1}). Therefore, the total number of iterations required to compute the inverse in Montgomery domain is $2k - n$. To compute the modular inverse in the integer domain it needs $2k$ iterations.

Number comparisons were used in $Alg2$ to compare the size of the bit vectors that represent elements in the field (the same way it was done in [9, 16, 15]) instead of the counter (δ) used in our algorithm. These *comparisons* are expensive in both fields and their time complexity is $O(log(n))$. The *comparison* limits a fast hardware implementation.

Figure 5.1 shows the number of iterations as a function of operand size required by $Alg1$ (UDMA) and $Alg2$ to compute the integer modular inverse (division) in $GF(p)$ and $GF(2^n)$. The size of the operands is in the range (160 to 512) bits.

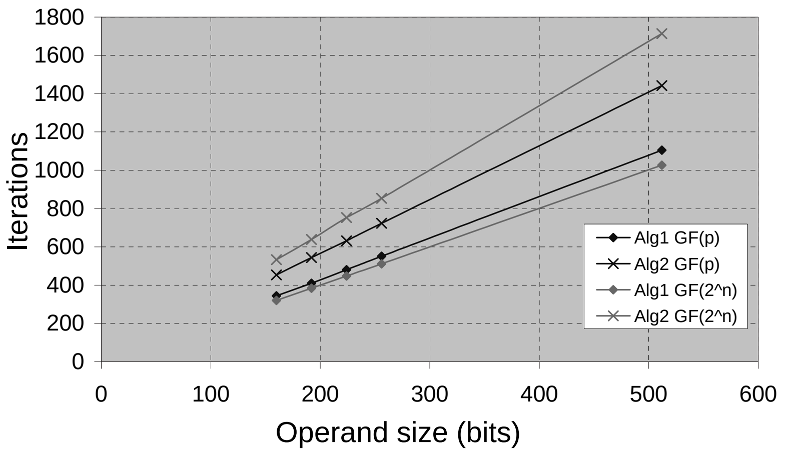


FIGURE 5.1: The number of iterations as a function of operand size required by $Alg1$ (UDMA) and $Alg2$ (presented in [10]) to compute the modular inverse in $GF(p)$ and $GF(2^n)$

From figure 5.1, we can see that $Alg1$ executes in about 25% less iterations than $Alg2$ when computing the inverse in the integer domain for $GF(p)$. For $GF(2^n)$, $Alg1$ has about 40% less iterations than $Alg2$ when computing the inverse in the integer domain.

Notice that the number of iterations for $Alg1$ in both fields increase linearly with the operand precision.

Table 5.1 shows the average number of additions used in $Alg1$ and $Alg2$. The $Gain = \frac{Alg2-Alg1}{Alg2}$ is also shown in the Table. The Table shows that $Alg1$ has up to 9% less additions than $Alg2$ when computing integer inverse in $GF(p)$, and 5% less additions in $GF(2^n)$.

When computing the inverse in Montgomery domain, $Alg2$ has less number of additions than $Alg1$. But, since the hardware implementation of $Alg1$ (see Chapter **??**)

$GF(p)$ n-bits	$Alg1$	$Alg2$	$Gain\%$
160	516	562	8.2
192	620	682	9
224	726	791	8.5
256	832	904	8
512	1660	1807	8

$GF(2^n)$ n-bits	$Alg1$	$Alg3$	$Gain\%$
160	643	671	4.2
192	762	811	6
224	895	931	3.9
256	1031	1072	3.8
512	2005	2113	5.1

TABLE 5.1: Average number of additions for $Alg1$ and $Alg2$ to compute the modular inverse in $GF(p)$ and $GF(2^n)$.

uses carry save adders, the additions are performed in almost constant time and will not increase the critical path delay as we will be seen in the next Section.

J. Goodman et al. presented in [16] a public-key processor that implements operations required for Elliptic Curve Cryptography (ECC) including modular inverse in $GF(2^n)$ only. It is stated in [16] that the inversion in $GF(2^n)$ takes on average 3.3 cycles for each bit. $Alg1$ needs a maximum of 2 iterations/bit and on average 1.5 iterations/bit to compute the modular inverse in $GF(2^n)$. The UMDM architecture performs each iteration of $Alg1$ in 2.5 clock cycles on average. Therefore, the $GF(2^n)$ inversion by $Alg1$ takes on average $1.5 \times 2.5 = 3.75$ cycles for each bit. But on the other hand, the critical path delay (clock period) of the datapath of the processor presented in [16] (let us call it $t_{G\mu P}$) is the delay of two full adders (t_{FA}) and two levels of multiplexers and an

AND gate ($t_{G\mu P} = 2t_{FA} + 2t_{mux} + t_{AND}$), compared to a critical path delay of only two full adders and one level of multiplexers for the UMDM datapath proposed in this work ($t_{UMDM} = 2t_{FA} + t_{mux}$). The critical path delay will affect the total computational time as will be seen in the next subsection.

The dual-field arithmetic unit proposed in [15] performs one addition in each clock cycle and the redundant adder in the architecture is used to convert from carry-save form to non-redundant representation, significantly increasing the number of clock cycles. So, the only way to compare our algorithm with [15] is to compare the number of additions, and in this case the number of additions reported in [15] is around 20 times greater than the number of additions in $Alg1$ [40].

The algorithm presented in [9] is a word-based algorithm that applies several strategies such as variable shifts. For this reason, we didn't consider it for comparison with $Alg1$.

$Alg1$ computes Montgomery modular multiplication in n cycles, which is consistent with several other designs [5, 6, 16].

5.2. Synthesis Results

The experimental data presented in this section were generated using Mentor Graphics CAD tools. The target technology was set to $AMI05\_fast\ auto$ (0.5 μm CMOS with hierarchy preserved) provided in the ASIC Design Kit (ADK) from the same company [50].

The unified modular divider/multiplier (UMDM) architecture was described in VHDL and simulated in ModelSim for functional correctness. It was synthesized using Leonardo synthesis tool for the mentioned technology. ADK provides a consistent environment for comparison between the designs, and a reasonable approximation of the system performance when using commercial ASIC technology.

Figure 5.2 shows the critical path delays (in *nano*-seconds) of the (UMDM) for the precision range from 128-bit to 512-bit. The delay at 128-bit is 11.77 *nsec*, at 256 is

12.51 *nsec*, and at 512-bit is around 12.8 *nsec*. form the Figure we can notice that the delay increases as the number of bits increase and it saturates at higher precision. This indicates that the critical path delay (clock period) of the UMDM became independent of the operands size at high precisions. This behavior comes out from the load that is applied to same high fan-out signals in the design.

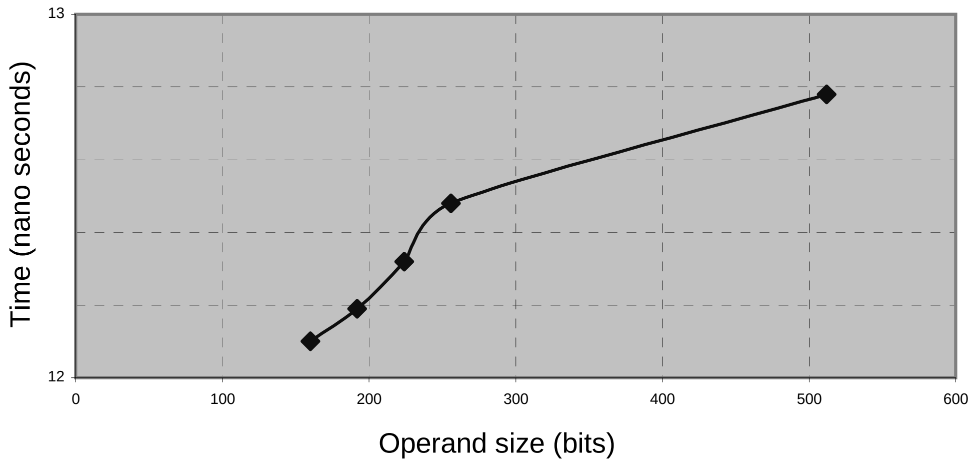


FIGURE 5.2: The critical path delay of the UMDM in *nano*-seconds (operand size from 160-512 bits).

The public-key processor presented by Goodman and et.al in [16] ($G\mu P$) runs at clock rate of 50 MHz (clock period = 20 *nsec*), and it is considered a good representative of this class of hardware designs. The divider/multiplier proposed in this work has a maximum clock period of 12.8 *nsec* at 512-bit operand size, which is about 1.6 times faster than the processor presented in [16]. Also, as we mentioned in the previous subsection, the $G\mu P$ takes 3.3 cycles/bit to perform a modular inverse in $GF(2^n)$ and this design takes 3.75 cycles/bit. Let the total average computation time of a given design be T_{design} which is given by:

$$T_{design} = (cycles/bit) * n * clock\ period.$$

Operand size (*bits*)	Area (gates)
128-bit	30403
160-bit	37059
192-bit	45513
224-bit	53075
256-bit	60629
512-bit	121070

TABLE 5.2: The Area of the UMDM Design in gates for different operand sizes

where n is the operand size in bits. At $n = 512$-bits, the total computation time of $G\mu P$ ($T_{G\mu P}$) is:

$$T_{G\mu P} = 3.3 * 512 * 20 \times 10^{-9} = 33.79\mu sec$$

and the total computation time for this design (T_{UMDM}) is:

$$T_{UMDM} = 3.75 * 512 * 12.8 \times 10^{-9} = 24.57\mu sec$$

Comparing $T_{G\mu P}$ and T_{UMDM}, we find that the UMDM computes division in $\frac{33.79-24.57}{33.79} = 27.3\%$ less total computation time than the $G\mu P$.

Table 5.2 shows the total number of gates for the UMDM design as a function of operand size. The area for the UMDM design was extracted from the experimental data presented in Table 5.2 as:

$$A_{UMDM} = 236.12 * n + 180 = \ O(n) \ gates$$

The architecture presented in [14] is dedicated to $GF(2^n)$ only, and it uses degree comparisons to keep track of the field polynomials, which results in longer critical path delay. The design has an area complexity of $O(n)$ too.

The integrating of Montgomery multiplication and modular division in one design will add extra gates when compared to a dedicated divider. In the design proposed in this work, Montgomery multiplication is computed in almost the same time and complexity of a separate multiplication unit. In addition to that, this design allows the ability to compute division in the same unit with the flexibility to choose the required finite field.

5.3. FPGA Synthesis Results for The Scalable Design

The scalable divider/multiplier design was synthesized for the Field Programmable Gate Arrays (FPGAs) $VertixII$ chip. The technology was set to $xc2vp50-7ff148$. The following two subsections present the area and the critical path delay results obtained for the design.

5.3.1. Area Results

Figure 5.3 shows the area synthesis results (in number of slices) of the scalable Unified Modular Divider/Multiplier (UMDM). The area is presented as function of the operands size (n) with different combinations of the datapath word sizes (w). The area results were obtained for the operands size in the range from 16 to 512 bits. The datapath word size was in the range from 16 to 256 bits. The reason why we did not use larger operand sizes, is because the machines we are using could not handle operand size greater than 512 bits.

From the Figure, we notice that the area increases linearly as the operand size increases. There is a little difference in the number of slices when using different datapath word sizes for the same operand size.

The area for the scalable UMDM design was extracted from the experimental data presented in Figure 5.3, approximately as:

$$A_{scUMDM} = 28 * n + 275 = \ O(n)$$

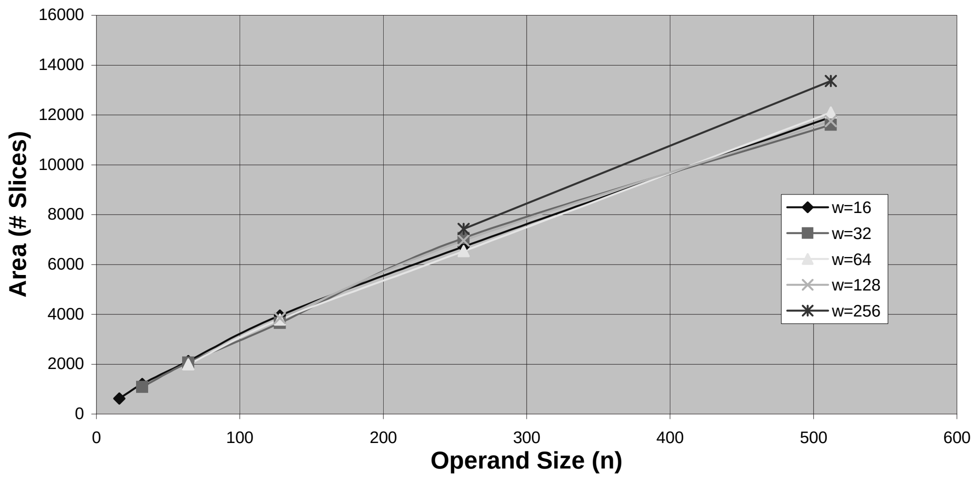


FIGURE 5.3: The Area (FPGA Technology) of the scalable UMDM in number of slices for combinations of operand size (n) form 16-512 bits, and datapath word size (w) from 16-256 bits.

5.3.2. Critical Path Delay Results

The same as in the area results, the experimental data for the critical path delay were obtained for the operands size (n) in the range from 16 to 512 bits, and the datapath word size (w) range from 16-256. Table 5.3 shows the critical path delay (clock period) for all the possible combinations of the operands size and the datapath word size. The "-" indicates the combination is not possible.

The operating frequency of the UMDM design can be found by taking the reciprocal of the clock period at any point. Form the Table, the lowest clock period (19.83 ns) is at $n = 16$ and $w = 16$, and so, the maximum operating frequency is around 50 MHz.

The question now is how to choose the best design points? or in other words, the (n, w) combinations that gives the lowest delay? By looking to Table 5.3, we notice that at a given operand size n, the minimum delay happens at the datapah word size $w = n$. For example, the best combination at the operand size $n = 256$, happens when the word size $w = 256$ also, with minimum delay equal to 28.4 *nano*-seconds.

	Datapath word size (w)				
Operands size (n)	16	32	64	128	256
16	19.83	-	-	-	-
32	24.55	22.13	-	-	-
64	25	26.55	24.7	-	-
128	32	31	27.9	25.4	-
256	34.7	37.3	34.3	31.9	28.4
512	47.15	38.71	38.5	37.4	35.4

TABLE 5.3: The critical path delay (clock period) of the scalable UMDM in *nano*-seconds for combinations of operand size (16-512 bits), and datapath word size from 16-256 bits

6. CONCLUSIONS AND FUTURE WORK.

6.1. Conclusion

This thesis has many contributions to the research in the areas of computer arithmetic algorithms (mainly in $GF(p)$ and $GF(2^n)$ finite fields) and scalable hardware designs for cryptographic applications. Below, is a brief description of the what we did in this research:

• In this thesis, we proposed an efficient unified modular division algorithm to compute modular division in $GF(p)$ and $GF(2^n)$ fields, and its hardware architecture. We also provided a mathematical proof for the algorithm. The algorithm uses a counter to keep track of the difference between two field elements, and this way eliminating the need for comparisons which are usually expensive and time-consuming. The hardware architecture that implements the algorithm is also proposed.

• Also, knowing that the Montgomery multiplication algorithm [5] is one of the most efficient ways to compute modular multiplication, we modified the unified Montgomery multiplication algorithm proposed by other members of our research group in [6] to have a control flow similar to the proposed division algorithm.

• We present a novel Unified modular Division/Multiplication Algorithm (UDMA) and its corresponding hardware architecture. The algorithm is a based on a Binary GCD algorithm for modular division in both fields, and on the Montgomery's Algorithm for modular multiplication. The UDMA computes the division in either $GF(p)$ or $GF(2^n)$ fields in an efficient way when compared with other algorithms. Because of using the counter when computing modular division, the iterations of the UDMA are less complex and faster than the iterations of other algorithms that use element comparisons. When compared with Montgomery multipliers, the proposed algorithm has the same number of iterations and complexity.

• The Unified Modular Divider/Multiplier (UMDM) design that implements the UDMA, efficiently supports all its operations and uses carry-save unified adders for reduced critical path delay.

• A scalable implementation of the UMDM was also described. The added scalability feature of the divisder/multipler allows a fixed-area datapath to handle operands of any size. Also, the word size of the datapath can be adjusted to meet the area and performance requirements.

• The proposed hardware design of the divider/multiplier was described in VHDL, and passed intensive simulation tests using ModelSim (Mentor Graphics tool). Then, it was synthesized for AMI $0.5\mu m$ CMOS technology and FPGA $VertixII$ chip ($xc2vp50-7ff148$ technology). The results are compared with other proposed dividers and multipliers.

• The experimental results show that the computation time of the proposed solution is competitive with other dedicated (limited) solutions, and the cost in area paid to have the integration of division and multiplication is not high. Multiplication is done almost as fast as in a dedicated multiplier with the added functionality of division and the flexibility to choose the required finite field.

6.2. Future work

There are many research contributions related to this study that can be investigated in the future. Some of them are:

• The testing process of the hardware implementation of the algorithm is tedious process. A methodology for developing testing modules is introduced in [51]. Including a self-testing block in the hardware design system will be beneficial and will reduce the time and effort for testing. One option could be is to perform modular division/Montgomery multiplication of hardwired numbers and compare the result with predefined values.

• Power dissipation study of the design is also needed in the context of power differential attack. This type of attack on a cryptographic system tries to deduce parameters of the system by observing system's power dissipation.

• The carry-save unified adders used in the hardware architecture are very efficient, and give very good results. Using other types of redundant adders should be studied, such as carry look-ahead adders [52].

• Integrating modular exponentiation with our divider/multiplier system, can be useful to some cryptographic applications that requires the three operations [53].

• The proposed divider/muliplier can be used as a basis for a cryptographic co-processor.

BIBLIOGRAPHY

1. T. ElGamal, "A public key cryptosystem and signature scheme based on discrete logarithms," *IEEE Trans. - Information Theory*, vol. IT-31, no. 4, pp. 469–472, July 1998.

2. M. E. Hellman and W. Diffie, "New directions on cryptography," *IEEE transactions on Information Theory*, vol. 22, pp. 644–654, November 1976.

3. L. Adleman, R. L. Rivest, and A. Shamir, "A method for obtaining digital signature and public-key cryptosystems," *Comm. of the ACM*, vol. 21, no. 2, pp. 120–126, February 1978.

4. National Institute for Standards and Technology, "Digital Signature Standard (DSS)," Tech. Rep. 168-2, FIPS PUB, January 2000.

5. P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, April 1985.

6. E. Savas, A. F. Tenca, and Ç. K. Koç, "A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$," in *Cryptographic Hardware and Embedded Systems — CHES 2000*, Ç. K. Koç and C. Paar, Eds. 2000, Lecture Notes in Computer Science, No. 1717, pp. 281–296, Springer, Berlin, Germany.

7. N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, January 1987.

8. A. J. Menezes, "Elliptic curve public key cryptosystems," 1993, Kluwer Academic Publishers, Bosten, MA.

9. A. A. Gutub, A. F. Tenca, E. Savas, and Ç. K. Koç , "Scalable and unified hardware to compute Montgomery inverse in $GF(P)$ and $GF(2^n)$," in *Cryptographic Hardware and Embedded Systems — CHES 2002*, B.S. Kaliski Jr. et al., Ed. 2003, Lecture Notes in Computer Science, No. 2523, pp. 484–499, Springer, Verlag Berlin Heidelberg 2003.

10. W. Trappe and L. C Washington, *Introduction to Cryptography with Coding Theory*, Prentice Hall, New Jersey, 2002.

11. E. Savas and Ç. K. Koç , "Architectures for unified field inversion with applications in elliptic curve cryptography," in *The 9th IEEE international conference on Electronic, Circuits and systems-ICECS 2002.*

12. D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Third edition, 1998.

13. I. F. Blake, G. Seroussi, and N. P. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, New York, 1999.

14. A. D. Daneshbeh and M. A. Hasan, "A unidirectional bit serial architecture for double-bases division over $GF(2^m)$," in *IEEE 16th Symposium on Computer Arithmetic.* 2003, IEEE Computer Society, Los Alamitos, California.

15. J. Wolkerstorfer, "Dual-field arithmetic unit for $gf(p)$ and $gf(2^n)$," in *Cryptographic Hardware and Embedded Systems — CHES 2002*, B.S. Kaliski Jr. et al., Ed. 2003, Lecture Notes in Computer Science, No. 2523, pp. 484–499, Springer, Verlag Berlin Heidelberg 2003.

16. J. Goodman and A. P. Chandrakasan, "An energy-efficient reconfigurable public-key cryptography processor," *IEEE Journal of solid-state circuits*, vol. 36, no. 11, pp. 1808–1820, November 2001.

17. J-H. Guo and C-L. Wang, "Systolic array implementation of Euclid's algorithm for inversion and division in $GF(2^m)$," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1161–1167, October 1998.

18. P. Choudhury and R. Barua, "Cellular automata based VLSI architecture for computing multiplication and inverses in $GF(2^m)$," in *Proceedings of the 7th IEEE International Conference on VLSI Design.* 1994, Calcutta, India.

19. H. Brunner, A. Curiger and M. Hofstetter, "On computing multiplicative inverse in $GF(2^m)$," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 1010–1015, August 1993.

20. G. Feng, " A VLSI architecture for fast inversion in $GF(2^m)$," *IEEE Transactions on Computers*, vol. 38, no. 10, pp. 1383–1386, October 1989.

21. M. A. Hasan, "Efficient computation of multiplicative inverse for cryptographic applications," in *IEEE 15th Symposium on Computer Arithmetic, Colorado.* 2001, IEEE Computer Society, Los Alamitos, California.

22. N. Takagi, "A hardware algorithm for modular division based on the extended Euclidean algorithm," *IEICE Trans. on Information and Systems*, vol. E79-D, no. 11, pp. 1518–1522, November 1996.

23. C. Wang, T. Truong, H. Shao, L. Deutsch, J. Omura and I. Reed, " A VLSI architecture for computing multiplications and inverses in $GF(2^m)$," *IEEE Transactions on Computers*, vol. C-34, no. 8, pp. 709–717, August 1985.

24. J-H. Guo and C-L. Wang, "Bit-serial systolic array implementation of Euclid's algorithm for inversion and division in $GF(2^m)$," *Proceedings of Technical Papers, International Symposium on VLSI Technology, Systems and Applications*, 1997.

25. M. Kovac, N. Ranganathan and M. Varanasi, " SIGMA: A VLSI systolic array implementation of Galois field $GF(2^m)$ based multiplication and division algorithm," *ieeetv*, vol. 1, no. 1, pp. 22–30, March 1993.

26. A. A.-A. Gutub, *New Hardware Algorithms and Designs for Montgomery Modular Inverse Computation in Galois Fields $GF(p)$ and $GF(2^n)$*, Ph.D. thesis, Oregon State University, Oregon,USA, June 2002.

27. N. Takagi, "A VLSI Algorithm for Modular Division Based on the Binary GCD algorithm," *IEICE Trans. fundamentals*, vol. E81-A, no. 5, pp. 724–728, May 1998.

28. R. Brent and H. Kung, "Systolic VLSI array for linear GCD computations," in *VLSI'83*, F. Anceau and E. Aas, Eds. 1983, pp. 145–154, Elsvier Science Publishers.

29. M. E. Kaihara and N. Takagi, "A VLSI Algorithm for Modular Multiplication/Division," in *IEEE 16th Symposium on Computer Arithmetic*. 2003, IEEE Computer Society, Los Alamitos, California.

30. A. F. Tenca, G. Todorov, and Ç. K. Koç, "High-radix design of a scalable modular multiplier," in *Cryptographic Hardware and Embedded Systems — CHES 2001*, Ç. K. Koç and C. Paar, Eds. 2001, Lecture Notes in Computer Science, No. 1717, pp. 189–206, Springer, Berlin, Germany.

31. C. D. Walter, "Space/time trade-offs for higher radix modular multiplication using repeated addition," *IEEE Transactions on computing*, vol. 46, no. 2, pp. 139–141, February 1997.

32. R. R. Taylor and S. C. Goldstein, "A high-performance flexible architecture for cryptography," in *Cryptographic Hardware and Embedded Systems*, C. Paar Ç K. Koç, Ed. 1999, number 1717 in Lecture Notes in Computer Science, pp. 231–245, Springer, Berlin, Germany.

33. L. A. Tawalbeh, "Radix-4 ASIC design of a scalable Montgomery modular multiplier using encoding techniques," M.S. thesis, Oregon State University, Oregon,USA, October 2002.

34. A. F. Tenca and L. A. Tawalbeh, "An efficient and scalable radix-4 modular multiplier design using recoding techniques," in *The Thirty-seventh Annual Asilomar Conference on Signals, Systems, and Computers*. November 9-12, 2003, vol. 2, pp. 1445–1450, IEEE Press, Pacific Grove, California.

35. A. Vandemeulebroecke and et al, "A new carry-free decision algorithm and its application to a single-chip 1024-bit RSA processor," *IEEE Journal of Solid-state Circuits*, vol. 25, no. 3, pp. 748–755, June 1990.

36. A. F. Tenca and Ç. K. Koç, "A word-based algorithm and scalable architecture for montgomery multiplication," in *Cryptographic Hardware and Embedded Systems — CHES 1999*, Ç. K. Koç and C. Paar, Eds. 1999, Lecture Notes in Computer Science, No. 1717, pp. 94–108, Springer, Berlin, Germany.

37. T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," in *Proceedings, 14th Symposium on Computer Arithmetic*, I. Koren and

P. Kornerup, Eds., Bath, England, April 14–16 1999, pp. 70–77, IEEE Computer Society, Los Alamitos, California.

38. G. Hachez and J-J Quisquater, "Montgomery exponentiation with no final subtractions: Improved results," in *Cryptographic Hardware and Embedded Systems — CHES 2000*, Ç. K. Koç and C. Paar, Eds. 2000, Lecture Notes in Computer Science, No. 1965, pp. 293–301, Springer, Berlin, Germany.

39. L. A. Tawalbeh, A. F. Tenca, S. Park, and Ç. K. Koç , "A dual-field modular division algorithm and architecture for application specific hardware," in *The Thirty-eighth Annual Asilomar Conference on Signals, Systems, and Computers.* November 7-10, 2004, IEEE Press, Pacific Grove, California.

40. A. F. Tenca and L. A. Tawalbeh, " An algorithm for unified modular division in $GF(p)$ and $GF(2^n)$ suitable for cryptographic hardware," *IEE Electronics Letters*, vol. 40, no. 5, pp. 304–306, March 2004.

41. M. Stan, A. Tenca, and M. Ercegovac, "Long and fast up/down counters," *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 722–734, July 1998.

42. J. Stein, "Computational problems associated with racah algebra," *Journal of Computational Physics*, vol. 1, no. 1, pp. 397–405, 1967.

43. A. F. Tenca, E. Savas, and Ç. K. Koç , "A design framework for scalable and unified multipliers in $GF(p)$ and $GF(2^n)$," *International Journal of Computer Research*, To appear 2004.

44. L.-S. Au and N. Burgess, "Unified radix-4 multiplier for $GF(p)$ and $GF(2^n)$," in *The IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'03)*, The Hague, The Netherlands, June 24-26 2003, pp. 226–232.

45. L. A. Tawalbeh and Alexandre. F. Tenca, "An algorithm and hardware architecture for integrated modular division and multiplication in $GF(p)$ and $GF(2^n)$," in *The IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP).* September 27-29, 2004, pp. 247–257, IEEE Computer Society Press, Los Alamitos, California.

46. M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann Publishers, California, 2004.

47. E. Savas, A. F. Tenca, and Ç. K. Koç , "Dual-field multiplier architecture for cryptographic applications," in *The Thirty-seventh Annual Asilomar Conference on Signals, Systems, and Computers.* November 9-12, 2003, pp. 374–378, IEEE Press, Pacific Grove, California.

48. E. Savas, A. F. Tenca, M. E. Ciftcibasi, and Ç. K. Koç , "Novel multiplier architectures for $GF(p)$ and $GF(2^n)$," *IEE Proceedings - Computers and Digital Techniques*, vol. 151, no. 2, pp. 147–160, March 2004.

49. A. Royo, J. Moran, and J. C. Lopez, "Design and implementation of a coprocessor for cryptography applications," in *European Design and Test Conference*, Paris, France, March 17-20 1997, pp. 213–217.

50. ASIC Design Kit. Mentor Graphics Co, "http://www.mentor.com /partners/hep/AsicDesignKit/dsheet/ami05databook.html," .

51. C. D. Walter, "Moduli for testing implementations of the rsa cryptosystem," in *IEEE 14th Symposium on Computer Arithmetic*. 1999, pp. 78–85, IEEE Computer Society, Los Alamitos, California.

52. A. A.-A. Gutub, A. F. Tenca and Ç. K. Koç , "Scalable VLSI architecture for GF(p) Montgomery modular inverse computation," in *IEEE Computer Society Annual Symposium on VLSI*. April 25-26, 2002, pp. 53–58, IEEE Computer Society Press, Los Alamitos, California.

53. A. J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, Florida, 1996.

54. D. M. Bressoud, *Factorization and Primality Testing*, Springer-Verlag, New York, 1989.

55. R. Crandall, *Prime Numbers: A Computational Perspective*, Springer-Verlag, New York, 2002.

56. Word IQ, "http://www.wordiq.com/definition/," .

APPENDICES

A THE GREATEST COMMON DIVISOR.

There are many important applications that depends on computing the greatest common divisor (gcd). Among theses applications are the modular arithmetic operations, specifically, computing modular inverse is heavily depends on the concept of the gcd [54].

The greatest Common Divisor of the two integers, a and b, $gcd(a, b)$ is the largest integer that evenly divides both of them, provided that $a,\ b\ \neq\ 0$ at the same time. Since every integer evenly divides zero, the above definition does not apply when a and b are both zero [12].

Below are some properties of the *gcd*:

$$gcd(0, 0) = 0 \tag{1.1}$$

Computing the *gcd* is a commutative operation:

$$gcd(a, b) = gcd(b, a) \tag{1.2}$$

Changing the sign of one of the integers, will not affect the value of the *gcd* as can be seen from the following equation:

$$gcd(a, b) = (-a, b) \tag{1.3}$$

Any integer divides itself, and it evenly divides zero, so:

$$gcd(a, 0) = |a| \tag{1.4}$$

If $gcd(a, b) = d$ and a divides the product bc, then $\frac{a}{d}$ divides c. Also, if m is any integer, then

$$gcd(ma, mb) = mgcd(a, b) \tag{1.5}$$

and

$$gcd(a + mb, b) = gcd(a, b). \tag{1.6}$$

If m is a nonzero common divisor of a and b, then:

$$gcd(\frac{a}{m}, \frac{b}{m}) = \frac{gcd(a,b)}{m} \tag{1.7}$$

The gcd of three numbers can be computed as:

$$gcd(a,b,c) = gcd(gcd(a,b),c) = gcd(a, gcd(b,c)). \tag{1.8}$$

One way to compute the gcd comes form the canonical factorization of integers. According to the "fundamental theorem of arithmetic" [12, 55], each positive integer a can be written in the form:

$$a = 2^{a_2}3^{a_3}5^{a_5}7^{a_7}... = \prod_P P^{a_P} \tag{1.9}$$

where P is a prime, and the $a_2, a_3, a_5, ...$ are nonnegative integers, including the value of zero.

To find the $gcd(a,b)$, we factor both of them to the canonical form as in equation 1.9, then we can write:

$$gcd(a,b) = \prod_P P^{min(a_P, b_P)} \tag{1.10}$$

Let us take the two integers $a = 500 = 2^2.5^3$ and $b = 2600 = 2^3.5^2.13^1$. The $gcd(a,b) = 2^{min(2,3)}.5^{min(3,2)}.13^{min(0,1)} = 2^2.5^2 = 100$.

Computing the gcd of two numbers by determining the prime factorizations is never done in practice, is too slow. Because it requires to determine the canonical factorization of a and b, and there is no known fast way to compute the prime factors of an integer[12].

A much more efficient method to compute the greatest common divisor without the need for factoring the integers was discovered more than 2000 years ago. This method is called the Euclidean algorithm [56].

An extended version of the Euclidean algorithm can also compute the integers u and v such that:

$$u.a + v.b = gcd(a, b) \tag{1.11}$$

This fact is used to compute the multiplicative inverse. More details about this subject and the Euclidean algorithm are provided in the next appendix.

B RELATED ALGORITHMS.

This appendix gives brief idea of some algorithms related to the research presented in this thesis.

B1. The Euclidean Algorithm.

The Euclidean algorithm, is an algorithm for finding the greatest common divisor of two non-negative integers a and b (the algorithm can be applied to the $\mid a \mid$ and $\mid b \mid$). The algorithm can also be defined for more general rings than just the integers, such as the binary extension fields ($GF(2^n)$) and the prime fields ($GF(p)$).

For a and b, there exist unique non-negative integers q and r such that:

and

$$a = qb + r \tag{2.1}$$

where $0 \leq r < b$. This is basic division operation where q is the quotient and r is the remainder [26]

The Euclidean Algorithm finds the $gcd(a, b)$ by repeated application of the division algorithm. The divisor is repeatedly divided by the remainder until the remainder of this operation is 0. The gcd is the last non-zero remainder in this algorithm.

```
**Function**: The Euclidean Algorithm
**Inputs**: Two Integers $a, b$
**Output**: The Greatest Common Divisor ($gcd(a, b)$)
IF $b = 0$, return $gcd(a, b) = b$
WHILE $b \neq 0$
  $r = a \ mod \ b$
  $a = b$
  $b = r$
return $GCD(a, b) = a$
```

The validity of the Euclidean Algorithm comes from equation 1.4, and the fact that [12]:

$$gcd(a, b) = gcd(a, a - qb) \tag{2.2}$$

The following example shows how to compute the $gcd(78, 45)$ by the Euclidean Algorithm:

$$78 = 45*1 + 33$$

$$45 = 33*1 + 12$$

$$33 = 12*2 + 9$$

$$12 = 9*1 + 3$$

$$9 = 3*3 + 0$$

The $gcd(78, 45) = 3$, which is the last non-zero remainder as mentioned earlier.

The Extended Euclidean Algorithm is an extension of the Euclidean Algorithm to compute the modular inverse. The inverse of an integer $a \in GF(p)$ with regard to the modulus p, is another integer t such that $a.t \equiv 1 \ mod \ p$. The inverse t exists if and only if a is relatively prime with p. In other words, $gcd(a, p) = 1$, which implies (from equation 1.11) the existence of the integers t and s such that:

$$t.a + s.p = 1 = gcd(a, p) \tag{2.3}$$

By reducing the equation $mod \ p$, the term $s.p$ will be zero, and we get $t.a \equiv 1 \ mod \ p$, and so t is the inverse of $a \ mod \ p$. The Extended Euclidean algorithm computes t efficiently. The same discussion can be used to find the inverse in the binary extension fields ($GF(2^n)$). For more details, the reader is referred to [13, 26].

B2. The Binary GCD Algorithm

A binary method to compute the *gcd* was proposed by J. Stein in 1967 [42]. It does not require division instruction as the Euclidean algorithm. It is based on the facts [27]:

1. If a is even and b is odd, then $gcd(a, b) = gcd(A/2, B)$, also $gcd(a, b) = gcd(a - b, b)$.

2. If a and b are both odd, then $a - b$ is even and $\mid a - b \mid < max(a, b)$.

The Binary GCD algorithm is described below:

```
Function: The Binary GCD Algorithm
Inputs: Two Integers $a, b$, $b$ is odd
Output: The Greatest Common Divisor ($gcd(a, b)$)
WHILE $a > 0$
  WHILE $a\ mod\ 2 = 0$        /* a is even */
    $a := a >> 1$
  END WHILE;
  IF $a \geq b$ THEN $a := a - b$
  ELSE $temp := a, a := b, b := temp$
  END IF;
END WHILE;
return $hcd(a, b) = b$;
```

The algorithm performs simple tests and 1-bit right shift operations ($a := a >> 1$). The comparison ($a \geq b$) is a time consuming step, and to avoid it, the authors in [28] modified the above algorithm and proposed the Plus-Minus algorithm, as described in Section B3.

B3. The Plus-Minus Algorithm

Brent and Kung proposed the Plus-Mins (PM) algorithm [28]. and they allowed the inputs a and b to be negative. The difference of the upper bounds exponents is used in the algorithm. In other words, let 2^{α} be the upper bound of $\mid a \mid$, and 2^{β} be the upper bound of $\mid b \mid$, then, the difference between the exponents is $\delta = \alpha - \beta$.

The PM algorithm is based on the following fact to compute the gcd in $GF(p)$ [27, 40]: If C and D are both odd, then 4 divides either $C + D$ or $C - D$. If the first is true, then $gcd(C, D) = gcd((C + D)/2, D) = gcd((C + D)/4, D)$, and $|(C + D)/4| \leq max(|C/2|, |D/2|)$. If 4 divides $(C-D)$ then $gcd(C, D) = gcd((C-D)/2, D) = gcd((C-D)/4, D)$, and $|(C - D)/4| \leq max(|C/2|, |D/2|)$. The PM algorithm is shown below:

```
**Function**: The Plus-Minus Algorithm
**Inputs**: Two Integers $a, b$
**Output**: The Greatest Common Divisor ($gcd(a, b)$)
WHILE $\mid a \mid > 0$
  WHILE $a \ mod \ 2 = 0$        /* $a$ is even */
    $a := a >> 1$
    $\delta := \delta - 1$
  END WHILE;
  IF $\delta < 0$ THEN
    $temp := a, a := b, b := temp$
    $\delta := -\delta$
  END IF;
  IF $(a + b) mod \ 4 = 0$ THEN $a := (a + b) >> 1$
  ELSE $a := (a - b) >> 1$
  END IF;
END WHILE
return $gcd(a, b) = b$;
```

B4. The Extended Binary GCD Algorithm for Modular Division

Takagi extended the Plus-Minus algorithm to compute the modular inverse by intertwining a procedure to find the multiplicative inverse along with the procedure to compute *gcd*. Further more, he extended it to compute modular division by modifying the multiplicative inverse procedure so it computes the quotient [27]. The computations were done in $GF(p)$ only.

The algorithm was denoted by the Extended Binary GCD algorithm, and for complete details about it, the reader is referred to [27].

The Extended Binary GCD algorithm formed the base for our research. We modified it and extended to work in both $GF(2^n)$ and $GF(p)$ with simpler control and data flows.

C NUMERICAL EXAMPLE FOR COMPUTING MODULAR INVERSE USING UDMA IN $GF(P)$.

Table C.1 shows the computation of the modular inverse by the UDMA presented in Chapter 3. In other words, the algorithm is computing $Z = \frac{1}{Y}\ mod\ p$. The computations are done in the prime field $GF(p)$, with ($X = 1$, $Y = 1039$, and $p = 2011$). The operation is set to division ($Op = div$) with the dividend (X=1) in order to compute the modular inverse.

The intermediate values of the variables: $delta, C, D, U$, and W are shown in the Table with the corresponding performed operation by the UDMA. The Table also shows which portion of the algorithm was carried out ($THEN$ or $ELSE$). Notice that the modular reduction step ($U := (U + u_0 * p) >> 1$) is performed every iteration.

Notice that $D = -1$, and in this case the result $Z = p - W = 2011 - 60 = 1951$ as explained in Chapter 3. To verify the result: $Z = \frac{1}{Y}\ mod\ p \Rightarrow\ ZY \equiv 1\ mod\ p$, and by substituting the corresponding values, we get: $1951.1039 \equiv 2027089 \equiv 1\ mod\ 2011$. So 1951 is the inverse of 1039 mod 2011.

Operations	δ	C	D	U	W
Initialization	0	1039	2011	1	0
$C := (C - D) >> 1$	0	-486	2011	1006	0
$C\ >>\ 1$	-1	-243	2011	503	0
$swap$,$C := (C + D) >> 1$	1	884	-243	1257	503
$C\ >>\ 1$	0	442	-243	1634	503
$C\ >>\ 1$	-1	221	-243	817	503
$swap$,$C := (C - D) >> 1$	1	-232	221	-157	817
$C\ >>\ 1$	0	-116	221	927	817
$C\ >>\ 1$	-1	-58	221	1469	817
$C\ >>\ 1$	-2	-29	221	1740	817
$swap$,$C := (C + D) >> 1$	2	96	-29	2284	1740
$C\ >>\ 1$	1	48	-29	1142	1740
$C\ >>\ 1$	0	24	-29	571	1740
$C\ >>\ 1$	-1	12	-29	1291	1740
$C\ >>\ 1$	-2	6	-29	1651	1740
$C\ >>\ 1$	-3	3	-29	1831	1740
$swap$,$C := (C - D) >> 1$	3	-16	3	960	1831
$C\ >>\ 1$	2	-8	3	480	1831
$C\ >>\ 1$	1	-4	3	240	1831
$C\ >>\ 1$	0	-2	3	120	1831
$C\ >>\ 1$	-1	-1	3	60	1831
$swap$,$C := (C - D) >> 1$	1	2	-1	1891	60
$C\ >>\ 1$	0	1	-1	1951	60
$C := (C + D) >> 1$	0	0	-1	2011	60

TABLE C.1: A modular inverse computation by the UDMA algorithm in $GF(p)$