Efficient Binary Field Multiplication on a VLIW DSP

Christian Sean Tergino

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

Patrick Schaumont
Wuchun Feng
Michael Hsiao

June 18, 2009
Blacksburg, VA

Keywords: Binary Field, Galois Field, GF, Multiplication, Digital Signal Processor, Very Long Instruction Word, Modular Multiplication, C64x+, Heterogeneous Multiprocessors,

Copyright 2009, Christian Sean Tergino

Efficient Binary Field Multiplication on a VLIW DSP

Christian Sean Tergino

ABSTRACT

Modern public-key cryptography relies extensively on modular multiplication with long operands. We investigate the opportunities to optimize this operation in a heterogeneous multiprocessing platform such as TI OMAP3530. By migrating the long-operand modular multiplication from a general-purpose ARM Cortex A8 to a specialized C64x+ VLIW DSP, we are able to exploit the XOR-Multiply instruction and the inherent parallelism of the DSP. The proposed multiplication utilizes Multi-Precision Binary Polynomial Multiplication with Unbalanced Exponent Modular Reduction. The resulting DSP implementation performs a $GF(2^{233})$ multiplication in less than 1.31us, which is over a seven times speed up when compared with the ARM implementation on the same chip. We present several strategies for different field sizes and field polynomials, and show that a 360MHz DSP easily outperforms the 500MHz ARM.

ACKNOWLEDGEMENTS

I would like to thank all of my instructors and advisors throughout my time at Virginia Tech. I would specifically like to thank my advisory committee members Michael Hsiao and Wuchun Feng. A special thanks goes to my advisor Patrick Schaumont, who has spent a good deal of time helping guide my research. I also would like to thank my research group and my fellow students for their support.

FOREWORD

This is an expanded version of a paper submitted to International Symposium on System-on-Chip on May 29th, 2009. It is currently under review.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION..1

CHAPTER 2: BACKGROUND..3

2.1 Prime Field vs. Binary Field..3

2.2 Binary Field..4

2.3 NIST Finite Fields..4

2.4 Montgomery's Algorithm..5

2.5 Karatsuba Multiplication..6

2.6 Unbalanced Exponent Modular Reduction..6

2.7 Texas Instruments Beagle Board and OMAP3530..8

2.8 Texas Instruments TMS320C64x+ DSP..9

2.9 Related Work..10

CHAPTER 3: PROPOSED METHOD..11

3.1 Multi-Precision Binary Polynomial Multiplication..11

3.2 Unbalanced Exponent Modular Reduction..13

CHAPTER 4: METHODOLOGY..15

CHAPTER 5: RESULTS..24

5.1 Tools and Environment..24

5.2 Execution Times on the OMAP3530..24

5.3 Cycle Count Compared To Other Implementations..28

5.4 Reduction Results..29

CHAPTER 6: FUTURE WORK..33

6.1 Improving Binary Field Multiplication..33

6.2 Implementing a Cryptosystem on the C64x+..34

6.3 Finite Field Multiplications on Other Processors..34

CHAPTER 7: CONCLUSIONS..35

REFERENCES..36

LIST OF FIGURES

FIGURE 1: TEXAS INSTRUMENTS BEAGLE BOARD……………………………………………7

FIGURE 2: OMAP3530 FUNCTIONAL BLOCK DIAGRAM……………………………………..8

FIGURE 3: BLOCK DIAGRAM OF THE C64X+ DSP…………………………………….……9

FIGURE 4: MULTI-PRECISION BINARY POLYNOMIAL MULTIPLICATION….…………………..11

FIGURE 5: EXECUTION DIAGRAM OF A MULTIPLICATION…………………………...…12

FIGURE 6: UNBALANCED EXPONENT MODULAR REDUCTION…………………….………14

FIGURE 7: DESIGN FLOW FOR DEVELOPING SOFTWARE FOR THE C64X+…………….……15

FIGURE 8: EXECUTION TIMES OF MULTIPLICATIONS……………………………..…..25

FIGURE 9: MULTIPLICATION EXECUTION CYCLES PER BIT…………………………...…26

FIGURE 10: MULTIPLICATION EXECUTION CYCLES PER BIT^2 ON C64X+……………….…..26

FIGURE 11: MULTIPLICATION EXECUTION CYCLES PER BIT^2 ON ARM………….……..…..27

FIGURE 12: MULTIPLICATION EXECUTION CYCLES PER BIT RAISED TO THE LOG$_2$3……..…27

FIGURE 13: AN ARM WITH MULGF VERSUS OUR IMPLEMENTATION…………………….…...29

FIGURE 14: CYCLES FOR UNBALANCED EXPONENT MODULAR REDUCTION………………...30

FIGURE 15: PERCENTAGE OF EXECUTION TIME NEEDED FOR REDUCTION…………….......31

LIST OF TABLES

TABLE I: NIST PRIME FIELD SIZES……………………………………………………….……5

TABLE II: NIST BINARY FIELD IRREDUCIBLE POLYNOMIALS ………..………………………..5

TABLE III: MODULAR MULTIPLICATION ON THE TI OMAP3530…………………………….25

TABLE IV: CYCLES FOR MODULAR MULTIPLICATION……………………………………….28

TABLE V: CYCLES FOR UNBALANCED EXPONENT MODULAR REDUCTION…….………..…...29

TABLE VI: CYCLE PERCENTAGE FOR REDUCTION…………………………………..……32

CHAPTER 1

INTRODUCTION

Modern mobile devices frequently make use of heterogeneous multi-processors, which allows them to execute a mix of multimedia applications and general-purpose information technology. The parallelism, as well as the architectural heterogeneity, ensures that these chips are very energy-efficient. For example, the TI OMAP3530 runs off less than 500mA at 5V [1]. However, this efficiency is only available when applications can optimally exploit the features of the architecture. This means that software applications must be developed with the specialized platform features in mind. Indeed, these features are typically ignored by general-purpose software compilers, and require either a clever, architecture-aware programmer, or else a specialized software library.

In this thesis we consider the efficient execution of modular multiplication of long operands. Modular multiplication is a cornerstone of public-key cryptography, and it is used in algorithms such as ECC, RSA and DSA. Most known software optimizations of cryptographic modular multiplications assume standard processor architectures, and focus on the algorithm. Some well-known examples are modular multiplications with Montgomery or Barrett reduction, or multiplication based on Karatsuba decomposition [2]. However, rather than developing new algorithms, we investigate the opportunities offered by a heterogeneous Multi-Processor System on Chip (MPSoC) architecture.

We investigate the implementation of modular multiplications on Texas Instrument's OMAP3530, which has an ARM Cortex A8 and a TI C64x+ DSP. We will use the C64x+ DSP in the OMAP3530 as a cryptographic accelerator. There are two arguments for this. The first is that this DSP is a Very Long Instruction Word (VLIW) architecture, which enables parallelism. The second is that the DSP has an XOR-Multiply

(XORMPY) instruction, which can accelerate binary field multiplications in $GF(2^m)$. Neither of these features is available on the ARM.

Because cryptographic operands typically are several hundreds of bits long, and because the C64x+ DSP is a 32-bit processor, we have to implement modular multiplications using multi-precision arithmetic, based on combining XORMPYs, XORs and shifts. We call this operation Multi-Precision Binary Polynomial Multiplication (MPBPM). The contribution of our work is an efficient implementation of MPBPM on the C64x+ DSP. We also present two efficient modular reduction techniques based on Unbalanced Exponent Modular Reduction (UEMR) [3]. While these algorithms are known, we are not aware of any Binary Field Multiplication implementations optimized for the C64x+ DSP. Our results show that the resulting binary field multiplication executes six times faster on the DSP when compared with the ARM. This comparison is made against an ARM which runs an optimized modular multiplication algorithm at a higher clock frequency.

The remainder of this thesis is organized as follows. Chapter 2 introduces Finite Fields and the TI OMAP3530. Chapter 3 presents our proposed methods for Multi-Precision Binary Polynomial Multiplication and Unbalanced Exponent Modular Reduction. Our methodology is explained in Chapter 4. Results are given in Chapter 5. Possible future work on this topic is discussed in Chapter 6, and conclusions are drawn in Chapter 7.

CHAPTER 2

BACKGROUND

This chapter reviews prime field and binary field arithmetic, Unbalanced Exponent Modular Reduction, Montgomery's algorithm, Karatsuba Multiplication, the TI Beagle Board, OMAP3530 and C64x+.

2.1 Prime Field vs. Binary Field

ECC, a very popular public-key cryptographic primitive, is implemented over $GF(p)$, a prime field, or $GF(2^m)$, a binary field. The National Institute of Standard and Technology (NIST) recommends five specific prime fields and five specific binary fields, with curves defined for each. $GF(p)$ defines a finite field of integers, where its elements are $\{0, 1, 2, 3, ..., p\text{-}2, p\text{-}1\}$ and p is a prime number. Every operation is performed modulo p. $GF(2^m)$ defines a Finite Field of binary polynomials, i.e. polynomials whose coefficients are each 0 or 1. The maximal term in a number in $GF(2^m)$ is x^{m-1}. Every operation in this field is performed modulo an irreducible polynomial, $f(x)$.

$GF(p)$ is popularly implemented in software while $GF(2^m)$ is usually implemented in hardware. $GF(2^m)$ multiplications are faster than $GF(p)$ in hardware because there are no carries, which results in a reduced critical path [4]. It is also quicker to compute the inverse in a binary field versus a prime field [2]. On the other hand, $GF(p)$ multiplications are faster than $GF(2^m)$ in software because processors have integer multipliers built into them. $GF(2^m)$ relies on binary polynomial multiplications and a large majority of processors do not have support for this. However, the C64x+ does have support for binary polynomial multiplication for Reed Solomon based error control coding [5]. This special purpose hardware can also be utilized by cryptographic algorithms by implementing MPBPM.

2.2 Binary Fields

Binary field numbers are within $GF(2^m)$ and are represented in the form

$$A(x) = a_0x^0 + a_1x^1 + a_2x^2 + \ldots + a_{m-2}x^{m-2} + a_{m-1}x^{m-1}, \quad (1)$$

where each coefficient $a_i = \{0, 1\}$. Registers can easily represent these numbers where each bit in a word is a coefficient. For each binary field, an irreducible polynomial $f(x)$ is defined:

$$f(x) = T(x) + x^m, \quad (2)$$

$$T(x) = x^0 + t_1x^1 + t_2x^2 + \ldots + t_{m-2}x^{m-2} + t_{m-1}x^{m-1}, \quad (3)$$

where every coefficient, $t_i = \{0, 1\}$. All operations in $GF(2^m)$ are performed modulo $f(x)$. Addition and subtraction are equivalent to performing a bitwise Exclusive-OR between the two operands. The first step of multiplication, Binary Polynomial Multiplication (BPM), is similar to integer multiplication. In the second step, partial products are added together, which is equivalent to performing bitwise Exclusive-ORs. Performing this BPM, $C(x) = A(x)B(x)$ yields

$$C(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{2m-3}x^{2m-3} + c_{2m-2}x^{2m-2}. \quad (4)$$

The product must be reduced to remain within $GF(2^m)$, which is done by performing $C(x)$ mod $f(x)$.

2.3 NIST Finite Fields

NIST recommends the usage of five prime fields and five binary fields. The NIST prime fields are $GF(p192)$, $GF(p224)$, $GF(p256)$, $GF(p384)$ and $GF(p521)$. The prime fields are referred to in the form of $GF(pn)$, where n is the number of bits needed to represent the prime modulo, p. Therefore, $GF(p192)$ means that the field has p elements, where p is a 192-bit prime number. The p values are shown for each NIST Prime Field in Table I.

The NIST Binary Fields are referred to in the form $GF(2^m)$, where m is the number of bits in the binary field. NIST has recommended an irreducible polynomial for each

binary field. These polynomials are chosen to make computations in that field very efficient. Each of these irreducible polynomials has either three or five terms and the biggest term in $T(x)$ is less than $x^{m/2}$. The NIST binary fields are $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$ and $GF(2^{571})$. Table II displays the irreducible polynomial for each

NIST Binary Field.

TABLE I. NIST PRIME FIELD SIZES

NIST Prime Field	Field Size
$GF(p192)$	$2^{192} - 2^{64} - 2^{0}$
$GF(p224)$	$2^{224} - 2^{96} + 2^{0}$
$GF(p256)$	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 2^{0}$
$GF(p384)$	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 2^{0}$
$GF(p521)$	$2^{521} - 2^{0}$

TABLE II. NIST BINARY FIELD IRREDUCIBLE POLYNOMIALS

NIST Binary Field	Irreducible Polynomial
$GF(2^{163})$	$x^{163} + x^{7} + x^{6} + x^{3} + x^{0}$
$GF(2^{233})$	$x^{233} + x^{74} + x^{0}$
$GF(2^{283})$	$x^{283} + x^{12} + x^{7} + x^{5} + x^{0}$
$GF(2^{409})$	$x^{409} + x^{87} + x^{0}$
$GF(2^{571})$	$x^{571} + x^{10} + x^{5} + x^{2} + x^{0}$

2.4 Montgomery's Algorithm

Montgomery's algorithm is commonly used to speed up reduction. In prime field, there exists a k such that $2^{k-1} < p < 2^{k}$. Let r be 2^{k} and let the p-residue of a number a be $\bar{a} = a \cdot r \ (mod\ p)$. The Montgomery product of two p-residues, $\bar{a}$ and $\bar{b}$, is $\bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \ (mod\ p) = c \cdot r \ \ (mod\ p)$. Thus, c can be calculated by performing Montgomery Reduction: $c = \ \bar{c} \cdot r^{-1} \ (mod\ p)$ [6].

For example, if p = 11 then r = 16 and r^{-1} = 9 because $r \cdot r^{-1} \ (mod\ 11) = \ 1$. Let $a = 5$ and $b = 4$. Then $\bar{a} = 5 \cdot 16 \ (mod\ 11) = 3$ and $\bar{b} = 4 \cdot 16 \ (mod\ 11) = 9$. Let $c = a \cdot b \ (mod\ p)$. Therefore $\bar{c} = 3 \cdot 9 \cdot 9 \ (mod\ 11) = 1$ and $c = 1 \cdot 9 \ (mod\ 11) = \ 9$.

Performing Montgomery Reduction is faster than regular reduction (dividing by p). Performing more multiplication with the same operands will reduce the total execution time because the p-residue needs to only be calculated once for each number [7]. This algorithm can similarly be applied to binary fields.

2.5 Karatsuba Multiplication

The Karatsuba method can be used to reduce the execution time for large operand multiplication. This method replaces some multiplications with additions and subtractions. With a base b, if an operand x has two digits, (x_1x_0), and operand y has two digits, (y_1y_0), then their product z is

$$z = (x_1y_1)b^2 + (x_0y_1 + x_1y_0)b + (x_0y_0). \tag{8}$$

Karatsuba showed that a multiplication can be eliminated from (8). The middle coefficient is equivalent to

$$x_1y_0 + x_0y_1 = (x_0y_0 + x_0y_1 + x_1y_0 + x_1y_1) - x_1y_1 - x_0y_0 \tag{9}$$

$$x_1y_0 + x_0y_1 = (x_0 + x_1)(y_0 + y_1) - x_1y_1 - x_0y_0. \tag{10}$$

Therefore (8) can be rewritten as

$$z = (x_1y_1)b^2 + ((x_0 + x_1)(y_0 + y_1) - x_1y_1 - x_0y_0)b + (x_0y_0). \tag{11}$$

For examples, if $b = 10^2$ and $x = 2345$ and $y = 6789$, then $x_1 = 23$, $x_0 = 45$, $y_1 = 67$ and $y_0 = 89$. Traditionally, to calculate $z = x \cdot y$, one would calculate $(45 \cdot 89) + (45 \cdot 67 + 23 \cdot 67)10^2 + (23 \cdot 67)10^4$, which has four multiplications. Using Karatsuba Multiplication, one would calculate $z_0 = 45 \cdot 89 = 4005$ and $z_2 = 23 \cdot 67 = 1541$ first. Next, z_1 would be calculated as follows: $z_1 = (45 + 23) \cdot (89 + 67) - z_2 - z_0 = 5062$. Hence, $z = z_0 + z_1b + z_2b^2 = 4005 + 5062 \cdot 10^2 + 1541 \cdot 10^4 = 15920205$, which is equal to $2345 \cdot 6789$.

(11) has two more additions and subtractions but one less multiplication than (8). Traditional multiplication is performed in $O(n^2)$ because every word must be multiplied by every other word. Karatsuba Multiplication can be performed in $O(n^{\log_2 3})$ [8].

2.6 Unbalanced Exponent Modular Reduction

Shen, Jin and You proposed using Unbalanced Exponent Modular Reduction over binary fields in [3]. $C(x)$ can be divided into two parts as shown in (5).

Since $$C(x) = C_l(x) + C_h(x)x^m \quad (5)$$

and $$T(x) \equiv x^m \bmod f(x) \quad (6)$$

then $$C(x) \equiv C_l(x) + C_h(x)T(x) \bmod f(x) \quad (7)$$

Both $C_l(x)$ and $C_h(x)$ are within $GF(2^m)$. Because of (6), (7) can be performed and repeated until $C_h(x)$ is zero, in which case $C(x)$ is completely reduced. When the largest nonzero term in $T(x)$ is x^k and $k < \frac{m}{2}$, then (7) need only be performed twice. This is the case for all NIST Binary Fields.

For example in $GF(2^4)$, $f(x) = x^4 + T(x)$ and $T(x) = x + 1$. Multiplying $(x^3 + x + 1)(x^2 + x)$ yields $C(x) = x^5 + x^4 + x^3 + x^2 + x^2 + x = x^5 + x^4 + x^3 + x$. Therefore $C_l(x) = x^3 + x$ and $C_h(x) = x + 1$. To reduce this product using UEMR, we multiply $C_h(x)$ by $T(x)$, which gives $(x + 1)(x + 1) = x^2 + x + x + 1 = x^2 + 1$. We take this and add it to $C_l(x)$, which yields $x^3 + x^2 + x + 1$. Since now $C_h(x)$ is zero, the product is completely reduced. If the $C_h(x)$ was not zero, this process would be repeated.

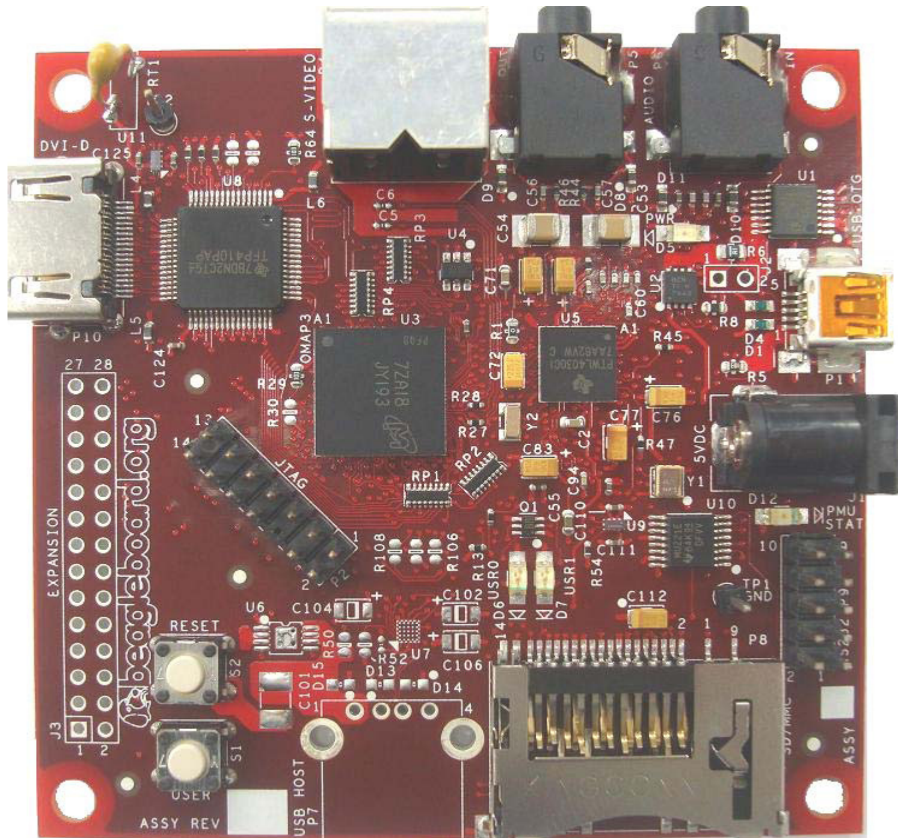

Figure 1. Texas Instruments Beagle Board [1]. Photo by Gerald Coley of BeagleBoard.org.

2.7 Texas Instruments Beagle Board and OMAP3530

The Beagle Board, shown in Figure 1, is a development board manufactured by Texas Instruments and available for purchase through DigiKey for $149. This board contains an OMAP3530, 128MB DDR RAM, 256 MB Flash, an SD Card Reader and a variety of I/O ports. The Beagle Board consumes less than 2500mW and can be powered by a USB port [1].

The TI OMAP3530 is a multiprocessor system in a Package-On-Package implementation built using 65nm technology [9]. It contains an ARM Cortex A8 and a TMS320C64x+ DSP [1]. Figure 2 depicts its Functional Block Diagram.

The ARM Cortex A8 is an applications processor based on the ARMv7 architecture and consumes less than 300mW [10]. The Cortex A8 supports Thumb-2 technology and is implemented with 16KB Instruction L1 Cache, 16KB Data L1 Cache and 256KB L2 Cache [9].

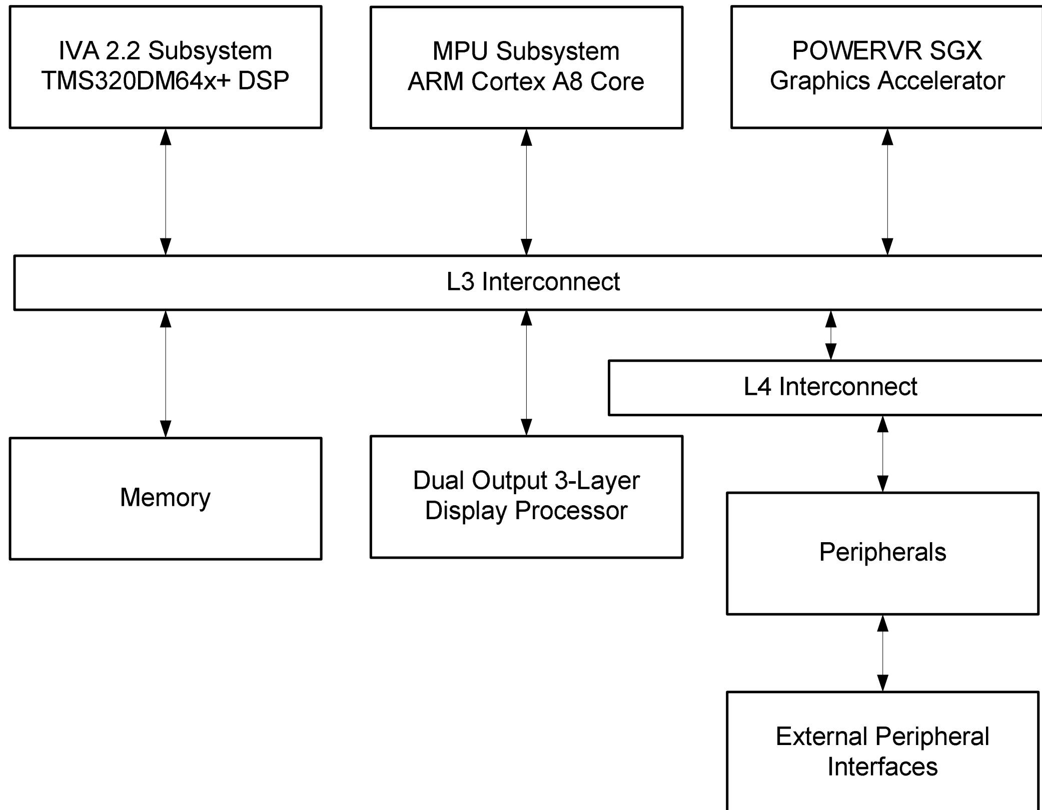

Figure 2. OMAP3530 Functional Block Diagram.

2.8 Texas Instruments TMS320C64x+ DSP

The C64x+ is a VLIW processor with 32KB of L1 Program Cache, 80KB of L1 Data Cache, and 96KB of L2 Cache [9]. It contains two identical data paths, each with four functional units. Each instruction is 32-bits and the processor can execute 8 instructions every cycle. Therefore every VLIW instruction fetch is 256-bits. The processor contains two register files, one for each data path. Each register file has 32 32-bit registers. The registers and data paths are shown in Figure 3. Each functional unit can optionally retrieve an operand from a register in the other data path. Shifts can be performed by the functional units .S1 and .S2. XORs, ANDs, and ORs can be performed by the functional units .S1, .S2, .D1, .D2, .L1 and .L2. Multiplications can be performed by .M1 and .M2 [11]. So in one cycle, the C64x+ can perform two shifts, four XORs and two multiplies. Alternatively, the C64x+ can perform six XOR instructions and two multiply instructions every cycle. Therefore an addition in $GF(2^m)$ can take just $\lceil m/(32 \cdot 6) \rceil$ clock cycles. Thus binary field addition can be performed faster on the C64x+ when compared to the ARM, since an ARM can only execute one 32-bit XOR every cycle.

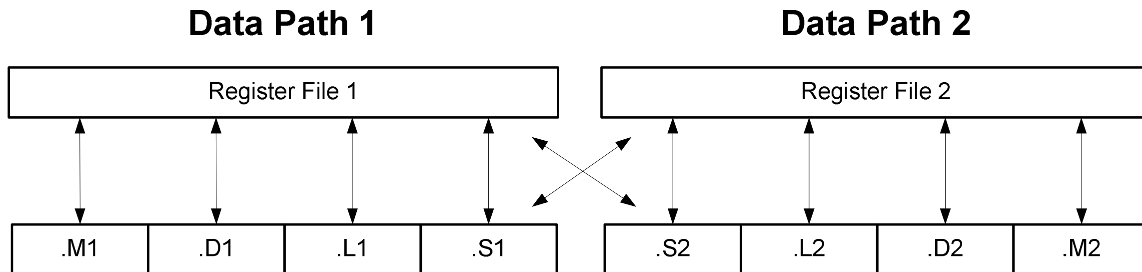

Figure 3. Block Diagram of the C64x+ DSP.

The C64x+ contains an XOR-Multiply (XORMPY) instruction, which is identical to a normal multiplication except that the partial products are XORed together instead of added. This is ideal for binary polynomial multiplication. The operand sizes for XORMPY are limited to 32-bits and 9-bits and the product size is limited to 32-bits.

2.9 Related Work

In [12], cryptosystems were implemented on the Texas Instruments TMS320C6201 DSP. The authors used Montgomery's algorithm for prime field multiplication on the DSP. This paper shows that RSA, DSA and ECC can run fast on a VLIW DSP.

In [13], the architecture of the TMS320C6201 was considered when enhancing an algorithm to perform modular multiplication. Gastaldo, Parodi and Zunino produced a 45% reduction in cycles for 2048-bit prime field multiplications using Montgomery's algorithm. Like [13], we analyze a DSP's architecture to help speed up modular multiplication. However, we instead use binary fields and Unbalanced Exponent Modular Reduction.

[14] shows that binary field multiplication can be sped up greatly by adding an instruction set extension, MULGF, to an ARM. MULGF performs 32-bit by 32-bit binary polynomial multiplication. However, adding this extension requires implementing the processor on an FPGA, which is not lower power, or designing a new ASIC, which takes significant time and resources. We will utilize a similar instruction, XORMPY, which is already part of the C64x+. Therefore our method requires no hardware design or implementation.

CHAPTER 3

PROPOSED MULTIPLICATION METHOD

In this chapter we present our proposed method for Multi-Precision Binary Polynomial Multiplication and our proposed implementations for Unbalanced Exponent Modular Reduction.

3.1 Multi-Precision Binary Polynomial Multiplication

We propose using the XORMPY instruction to perform Multi-Precision Binary Polynomial Multiplications. XORMPY puts its result in a 32-bit register. Since its operands are limited to 32-bits and 9-bits, the results extends out by 8-bits (32+9-1). Since

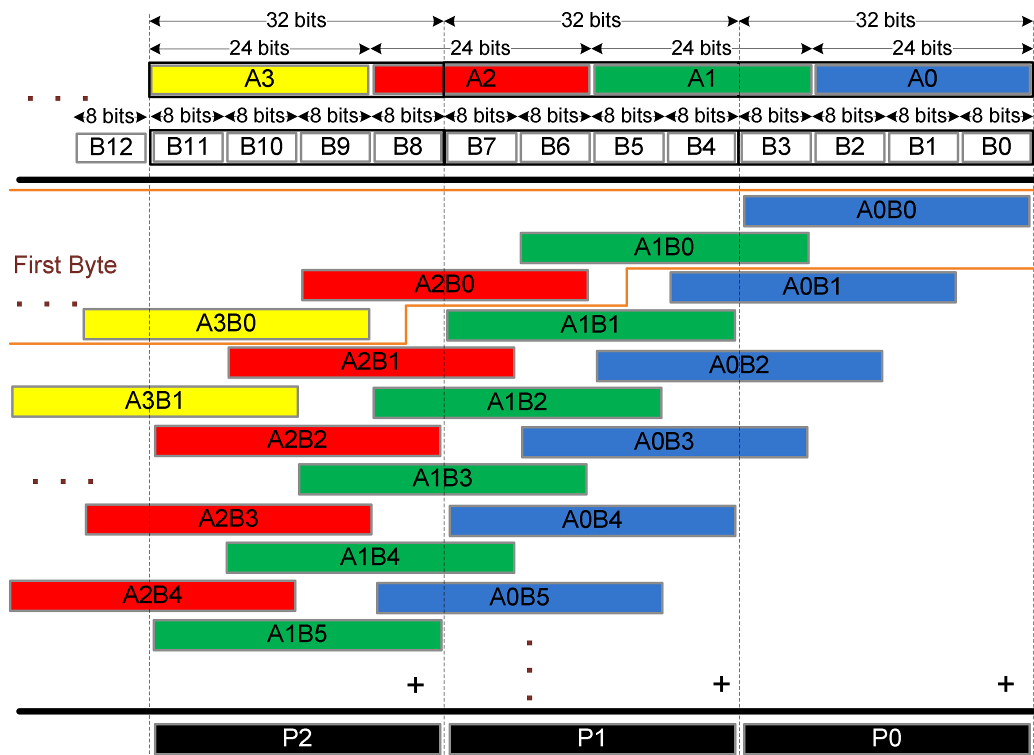

Figure 4. Multi-Precision Binary Polynomial Multiplication: Partial products from 24-bit by 8-bit XORMPY are XORed into 32-bit products.

we need all bits in the result, the sum of the number of bits in the operands must be less than or equal to 33 (not 32 because there are no carries). To keep everything byte-aligned, we propose performing 24-bit by 8-bit XORMPYs. Figure 4 shows an operand, "A", being multiplied by an operand, "B." Operand "A" must be shifted into 24-bit subwords. Each subword can then be multiplied by each byte of the "B" operand. This will produce partial products which can then be XORed together to form a complete product, "P". Figure 4 shows the formation of the partial products and complete product.

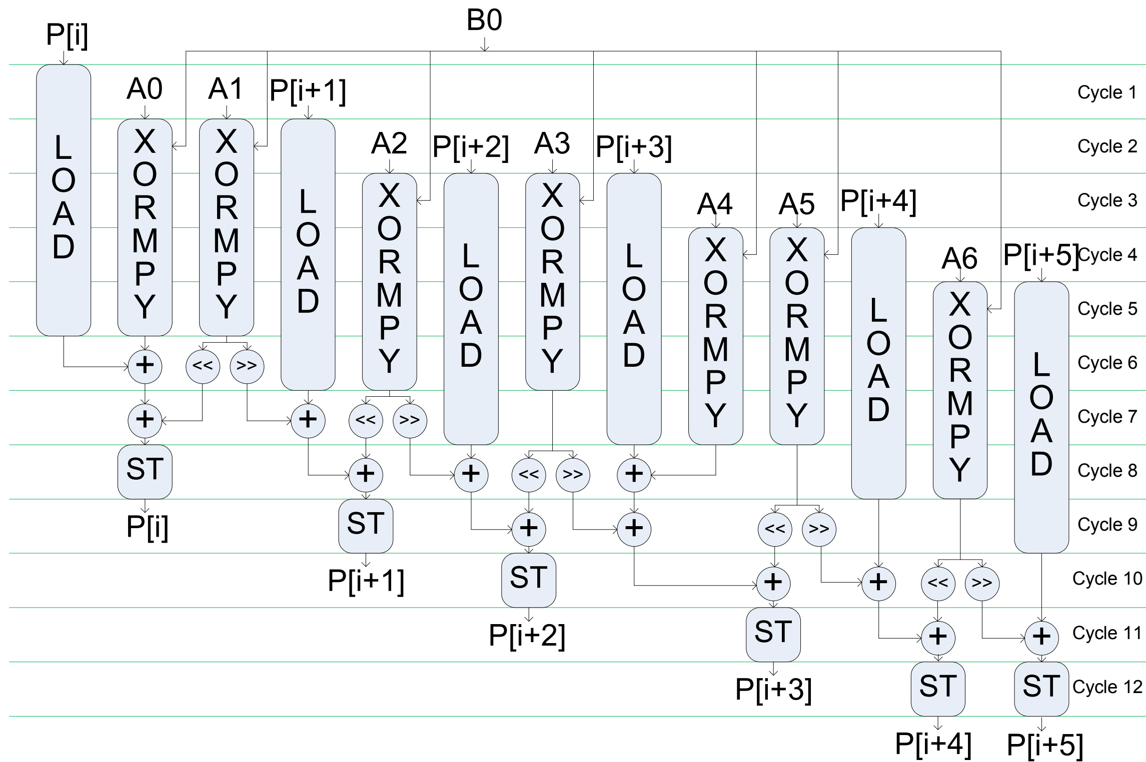

Figure 5. Execution diagram of a multiplication between a 24-bit word of operand A and a byte of operand B in $GF(2^{163})$

Performing the XORMPY operations and then shifting and XORing the result is the most computationally intensive part of this algorithm. Given the C64x+'s parallelized architecture, this series of instructions can be computed quickly. The DSP can perform two XORMPYs, two shifts along with four XOR operations every cycle. Figure 5 shows the execution diagram of the formation of the partial products using the "First Byte" portrayed in Figure 4. Figure 5 details the dependency and ability to take advantage of the parallelized architecture of the C64x+. The rows represent cycles and the columns show concurrent operations. Even though only two XORMPYs can be dispatched every cycle,

more than two XORMPYs can be executed concurrently. For example in Cycle 5, one XORMPY is starting to execute, but six other XORMPY instructions are also being executed. In Cycle 8, three XOR instructions, two Shift instructions and one Store instruction are all executed in parallel.

3.2 Unbalanced Exponent Modular Reduction

After the full polynomial product is formed, it must be reduced into $GF(2^m)$. We propose using Unbalanced Exponent Modular Reduction [3].

The irreducible polynomial is usually known at compile time, and based on this we propose using one of two methods for Unbalanced Exponent Modular Reduction. If three of four or more terms in $T(x)$ are within 9 bits of each other, then it is beneficial to use XORMPYs, shifts and XORs to reduce the product. Else, it is more efficient to shift and add for reduction.

If most of terms in $T(x)$ are within a 9-bit window, then these terms can fit in the 9-bit operand of XORMPY. Therefore, XORMPY can be used by multiplying $T(x)$ by $C_h(x)$ using 24-bits at a time. This is similar in time and resources to multiplying one byte of operand B by all 24-bit words of operand A. If there is a term that is not within this window, shifts and adds can be used implement that bit's multiplication.

For example, in the case of $GF(2^{163})$, the irreducible polynomial is: $f(x) = x^{163} + x^7 + x^6 + x^3 + x^0$. So $T(x) = x^7 + x^6 + x^3 + x^0$ (0xC9). Performing the first iteration of reduction requires $T(x)$ be multiplied with seven 24-bit words. The next and final iteration requires $T(x)$ be multiplied with one 24-bit word. The left diagram of Figure 6 shows a 325-bit product being reduced to $GF(2^{163})$.

When most of the set bits are not in a 9-bit window of the irreducible polynomial, it is more effective to manually multiply the upper bits. Instead of calling XORMPY, for coefficient t_i which is 1 in $T(x)$, shift all bits m and greater to the right m-i bits. For each term produced, XOR them together and with the original bits below m.

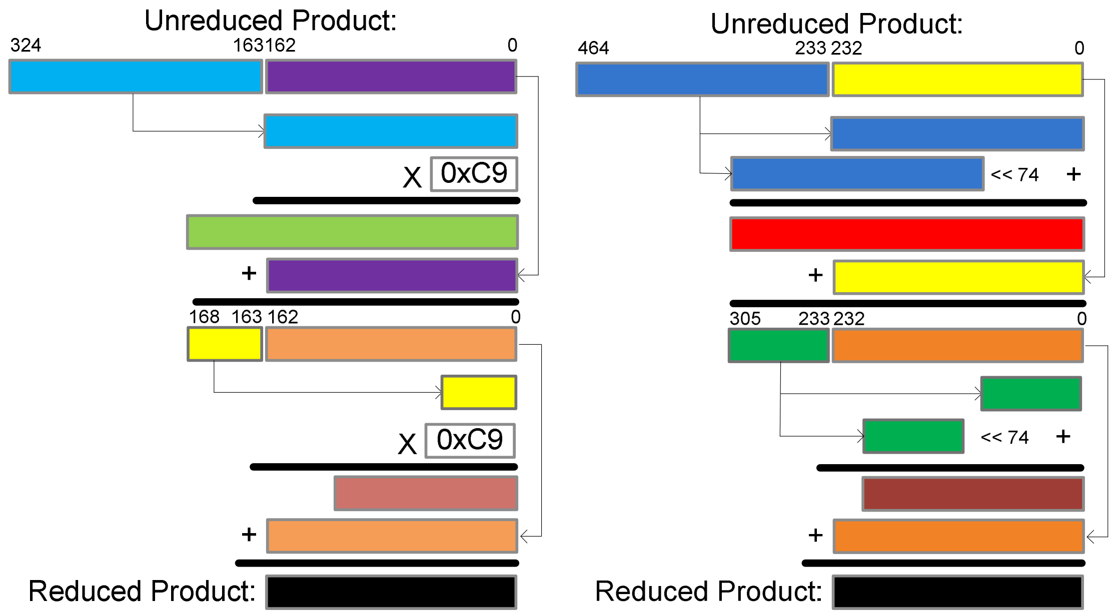

Figure 6. Unbalanced Exponent Modular Reduction. Left: Reduction over $GF(2^{163})$ using MPBPM. Right: Reduction over $GF(2^{233})$ using shifts and adds.

For $GF(2^{233})$, the irreducible polynomial, $f(x)$, is $x^{233} + x^{74} + x^0$. To reduce, we shift all bits above bit 232 to the right 233, then XOR them with the bits occupying bits 232-0. We XOR that result with $C_h(x)$ shifted to the right 233-74 = 159 bits. Since the result extends out over 233 bits, these steps must be repeated once more. This is shown in the right diagram of Figure 6.

CHAPTER 4

METHODOLOGY

We used the TI C6x C Compiler to build our program. All of our coding was in C. The compiler has built-in functionality which recognizes when the function "_xormpy()" is written in a C program it refers to the XORMPY instruction. Given the inherent parallelism of the DSP, the compiler attempts to make the best use of its hardware when compiling the source code. Optimizing code for a processor with eight parallel functional units that can all access the same registers is complicated. The compiler can produce very different binary files by just switching the order of a few C statements, even if the resulting executable will have the exact same functionality. Therefore our design cycle, depicted in Figure 7, had some added steps to try to get the most out of the compiler.

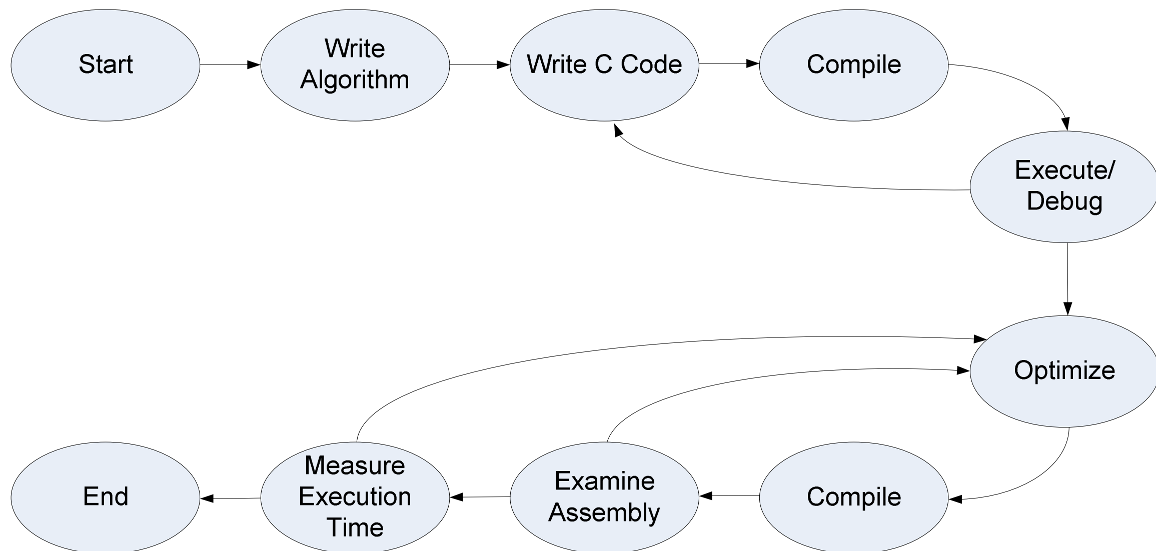

Figure 7. Design flow for developing software for the C64x+.

First, we developed the algorithm and pseudo code. Next, we wrote the C code and compiled it. Then we went through a few cycles of debugging and fixing the C Code. Once the program was functioning properly, we worked on optimizing the resulting

binary, by optimizing the C code. Some techniques we used to help reduce execution time are listed below:

- Put C statements which could be executed in parallel, adjacent to each other
- Unroll and roll loops
- Adjust the number of temporary variables
- Do not access registers modified by instructions with delay slots soon after dispatched

After making one of the above changes, we would compile the modified program and compare its assembly listing file to the previous iteration's assembly listing. If the new listing file looked more optimized, we would work on further optimizing this new C file. Else, we would go back to the previous iteration's C file and try different optimizations. We considered an assembly listing more optimized if the number of cycles needed decreased from the previous listing file.

The following C Code displays the main loop for a $GF(2^{163})$ multiplication. The compiled assembly listing of this code is listed immediately following the C Code. Each line in the assembly listing starting with "||" indicates it is being dispatched in parallel with the preceding instruction. Counting every instruction not preceded by "||", gives a good estimate for the number of cycles needed to perform that group of instructions. Therefore, assuming there are no cache misses, each iteration in this loop will take 96 cycles. There will be a very limited number of cache misses because the L1 Data Cache is 80KB and the data required for this multiplication is much less than 1KB. There are 227 total instructions being executed in these 96 cycles, including 23 cycles of NOPs (No-Operation instructions). Therefore, each cycle an average of 2.13 instructions are being dispatched in parallel, excluding NOPs. This loop will run for 5 iterations, so it takes a total of $96 \cdot 5 = 480$ cycles.

The subsequent listing shows the same C Code with only two modifications. First, instead of using one variable to compute the partial products, seven variables are used (one for each partial product computed for each byte of operand "b"). Second, all

XORMPY instruction calls for the same byte of operand "b", are written on consecutive lines, and all XOR and shift instruction calls for that byte are also written consecutively. Even though this has the exact same functionality as the previous C code, compiling this gives much different results. The resulting assembly listing is given after the optimized C code. If there are no cache misses, each iteration of this loop will take 43 cycles. Since there are a total of 165 instructions (with no NOPs) being executed in these 43 cycles, each cycle an average of 3.84 instructions are being dispatched. Since this is executed for five iterations, this section of the code requires $43 \cdot 5 = 215$ cycles. This is a speed up of 2.23 when compared with the unoptimized C code.

This drastic difference in cycles can be surprising since there was no algorithmic change. However, this shows the importance of being architecturally-aware when writing code for a VLIW processor.

Unoptimized C Code Snippet from $GF(2^{163})$ Multi-Precision Binary Polynomial Multiplication

```
99  for(i=0; i<5; i++)
100 {
101
102         b = 0xFF&opb[i];
103
104         p = _xormpy(shifted[0], b);
105         final[i] ^= p;
106
107         p = _xormpy(shifted[1], b);
108         final[i] ^= (p << 24);
109         final[i+1] ^= (p >> 8);
110
111         p = _xormpy(shifted[2], b);
112         final[i+1] ^= (p << 16);
113         final[i+2] ^= (p >> 16);
114
115
116         p = _xormpy(shifted[3], b);
117         final[i+2] ^= (p << 8);
118         final[i+3] ^= (p >> 24);
119
120         p = _xormpy(shifted[4], b);
121         final[i+3] ^= p;
122
123         p = _xormpy(shifted[5], b);
124         final[i+3] ^= (p << 24);
125         final[i+4] ^= (p >> 8);
126
127         p = _xormpy(shifted[6], b);
128         final[i+4] ^= (p << 16);
129         final[i+5] ^= (p >> 16);
130
131
132
133         b = (0xFF00&opb[i]) >> 8;
134
135         p = _xormpy(shifted[0], b);
136         final[i] ^= (p << 8);
137         final[i+1] ^= (p >> 24);
138
139         p = _xormpy(shifted[1], b);
140         final[i+1] ^= (p);
141
142         p = _xormpy(shifted[2], b);
143         final[i+1] ^= (p << 24);
144         final[i+2] ^= (p >> 8);
145
146         p = _xormpy(shifted[3], b);
147         final[i+2] ^= (p << 16);
148         final[i+3] ^= (p >> 16);
149
150         p = _xormpy(shifted[4], b);
151         final[i+3] ^= (p << 8);
152         final[i+4] ^= (p >> 24);
153
154         p = _xormpy(shifted[5], b);
155         final[i+4] ^= (p);
156
157         p = _xormpy(shifted[6], b);
158         final[i+4] ^= (p << 24);
159         final[i+5] ^= (p >> 8);
160
161
162
163         b = (0xFF0000&opb[i]) >> 16;
164
165
166         p = _xormpy(shifted[0], b);
167         final[i] ^= (p << 16);
168         final[i+1] ^= (p >> 16);
169
170         p = _xormpy(shifted[1], b);
171         final[i+1] ^= (p << 8);
172         final[i+2] ^= (p >> 24);
173
174         p = _xormpy(shifted[2], b);
175         final[i+2] ^= (p);
176
177         p = _xormpy(shifted[3], b);
178         final[i+2] ^= (p << 24);
179         final[i+3] ^= (p >> 8);
180
181         p = _xormpy(shifted[4], b);
182         final[i+3] ^= (p << 16);
183         final[i+4] ^= (p >> 16);
184
185         p = _xormpy(shifted[5], b);
186         final[i+4] ^= (p << 8);
187         final[i+5] ^= (p >> 24);
188
189         p = _xormpy(shifted[6], b);
190         final[i+5] ^= p;
191
192
193
194         b = (0xFF000000&opb[i])>>24;
195
196         p = _xormpy(shifted[0], b);
197         final[i] ^= (p << 24);
198         final[i+1] ^= (p >> 8);
199
200         p = _xormpy(shifted[1], b);
201         final[i+1] ^= (p << 16);
202         final[i+2] ^= (p >> 16);
203
204         p = _xormpy(shifted[2], b);
205         final[i+2] ^= (p << 8);
206         final[i+3] ^= (p >> 24);
207
208         p = _xormpy(shifted[3], b);
209         final[i+3] ^= (p);
210
211         p = _xormpy(shifted[4], b);
212         final[i+3] ^= (p << 24);
213         final[i+4] ^= (p >> 8);
214
215         p = _xormpy(shifted[5], b);
216         final[i+4] ^= (p << 16);
217         final[i+5] ^= (p >> 16);
218
219         p = _xormpy(shifted[6], b);
220         final[i+5] ^= (p << 8);
221         final[i+6] ^= (p >> 24);
222
223
224 }
```

Assembly Output from Compiling Unoptimized C Code Snippet from $GF(2^{163})$ Multi-Precision Binary Polynomial Multiplication

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*Loop source line                 : 99
;*Loop opening brace source line   : 100
;*Loop closing brace source line   : 224
;*      Known Minimum Trip Count         : 5
;*      Known Maximum Trip Count         : 5
;*      Known Max Trip Count Factor      : 5
;*Loop Carried Dependency Bound() : 69
;*      Unpartitioned Resource Bound     : 34
;*      Partitioned Resource Bound(*)    : 40
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                    21       26
;*      .D units                    40*      27
;*      .M units                    11       17
;*      .X cross paths              28       21
;*      .T address paths            40*      39
;*Long read paths              0        0
;*Long write paths             0        0
;*Logical  ops (.LS)           0        0
;*      Addition ops (.LSD)         29       34
;*      Bound(.L .S .LS)            11       13
;*      Bound(.L .S .D .LS .LSD)    30       29
;*
;*----------------------------------------------------------------------------*
$C$L1: ; PIPED LOOP PROLOG
;** --------------------------------------------------------------------------*
        LDW     .D2T2   *B16,B4         ; |102| <0,0>
        NOP             4
        EXTU    .S2     B4,24,24,B8     ; |102| <0,5>

        LDW     .D2T1   *+SP(32),A21    ; |127| <0,6>
||      XORMPY  .M2     B19,B8,B4       ; |109| <0,6>

        XORMPY  .M2     B6,B8,B7        ; |118| <0,7>

        LDW     .D1T1   *++A18,A5       ; |105| <0,8>
||      LDW     .D2T2   *B5++,B7        ; |105| <0,8>
||      XORMPY  .M2     B18,B8,B21      ; |111| <0,8>

        LDW     .D2T1   *+B5(4),A3      ; |113| <0,9>
||      LDW     .D1T2   *+A18(8),B7     ; |113| <0,9>
||      XORMPY  .M2     B20,B8,B9       ; |105| <0,9>

        NOP             1

        XORMPY  .M1X    A21,B8,A3       ; |129| <0,11>
||      SHL     .S2     B7,8,B22        ; |120| <0,11>

        LDW     .D2T2   *+B5(16),B7     ; |125| <0,12>
||      XORMPY  .M1X    A20,B8,A8       ; |123| <0,12>
||      SHRU    .S2     B7,24,B23       ; |118| <0,12>

        LDW     .D2T1   *+B5(12),A4     ; |125| <0,13>
||      XORMPY  .M2     B17,B8,B4       ; |121| <0,13>
||      XOR     .L2     B9,B7,B9        ; |105| <0,13>
||      SHRU    .S2     B21,16,B8       ; |113| <0,13>

        STW     .D2T2   B9,*-B5(4)      ; |107| <0,14>
||      XOR     .L2     B7,B23,B8       ; |118| <0,14>
||      XOR     .S2X    A3,B8,B7        ; |113| <0,14>
||      SHRU    .S1X    B4,8,A4         ; |109| <0,14>

        SHL     .S1X    B4,24,A4        ; |111| <0,15>
||      XOR     .L1     A5,A4,A6        ; |109| <0,15>
||      MV      .L2     B8,B23          ; |120| <0,15>
||      XOR     .S2     B7,B22,B22      ; |120| <0,15>
||      STW     .D2T2   B7,*+B5(4)      ; |116| <0,15>

        SHL     .S1     A8,24,A7        ; |124| <0,16>
||      SHRU    .S2X    A3,16,B9        ; |129| <0,16>
||      XOR     .L1X    B9,A4,A4        ; |111| <0,16>
||      STNDW   .D2T2   B23:B22,*+B5(4); |120| <0,16>
||      MV      .D1     A6,A5           ; |111| <0,16>

        SHRU    .S1     A8,8,A3         ; |125| <0,17>
||      SHL     .S2X    A3,16,B22       ; |129| <0,17>
||      XOR     .L2     B8,B4,B4        ; |121| <0,17>
||      STNDW   .D2T1   A5:A4,*-B5(4)   ; |111| <0,17>

        SHL     .S2     B21,16,B4       ; |112| <0,18>
||      XOR     .L1     A4,A3,A3        ; |125| <0,18>
||      XOR     .L2X    B4,A7,B8        ; |124| <0,18>
||      STW     .D2T2   B4,*+B5(8)      ; |123| <0,18>

        XOR     .L2X    A6,B4,B4        ; |112| <0,19>
||      STW     .D1T2   B8,*+A18(8)     ; |124| <0,19>

        MV      .L2X    A3,B4           ; |125| <0,20>
||      STW     .D2T2   B4,*B5          ; |112| <0,20>

        XOR     .L2     B7,B9,B9        ; |129| <0,21>
||      XOR     .S2X    A3,B22,B8       ; |129| <0,21>
||      STW     .D2T2   B4,*+B5(12)     ; |127| <0,21>

        STNDW   .D2T2   B9:B8,*+B5(12) ; |129| <0,22>
        LDW     .D2T2   *B16,B4         ; |133| <0,23>
        NOP             4
        EXTU    .S2     B4,16,24,B7     ; |133| <0,28>
        XORMPY  .M2     B20,B7,B4       ; |137| <0,29>

        LDW     .D1T1   *+A18(12),A7    ; |152| <0,30>
||      XORMPY  .M2     B17,B7,B8       ; |150| <0,30>

        LDW     .D1T2   *A18,B7         ; |137| <0,31>
||      XORMPY  .M1X    A19,B7,A4       ; |148| <0,31>

        LDW     .D2T2   *-B5(4),B4      ; |139| <0,32>
||      XORMPY  .M1X    A21,B7,A3       ; |159| <0,32>

        XORMPY  .M1X    A20,B7,A6       ; |155| <0,33>
||      SHL     .S2     B4,8,B9         ; |139| <0,33>

        LDW     .D1T1   *+A18(4),A6     ; |144| <0,34>
||      XORMPY  .M2     B18,B7,B9       ; |142| <0,34>
||      SHRU    .S2     B8,24,B21       ; |152| <0,34>

        LDW     .D1T1   *+A18(16),A5    ; |152| <0,35>
||      XORMPY  .M2     B19,B7,B21      ; |140| <0,35>
||      SHRU    .S2     B4,24,B4        ; |137| <0,35>
||      XOR     .L2X    A7,B21,B26      ; |152| <0,35>

        LDW     .D1T1   *+A18(8),A5     ; |144| <0,36>
||      XOR     .L2     B7,B4,B7        ; |137| <0,36>
||      SHL     .S2X    A4,16,B22       ; |150| <0,36>

        XOR     .L2     B4,B9,B24       ; |139| <0,37>
||      SHL     .S1     A3,24,A5        ; |159| <0,37>
||      SHRU    .S2X    A3,8,B23        ; |159| <0,37>
||      MV      .D2     B7,B25          ; |139| <0,37>

        XOR     .L1X    A7,B21,A3       ; |152| <0,38>
||      XOR     .L2X    B26,A6,B4       ; |155| <0,38>
||      STNDW   .D2T2   B25:B24,*-B5(4); |139| <0,38>

        SHRU    .S1X    B9,8,A3         ; |144| <0,39>
||      XOR     .L2     B7,B21,B4       ; |140| <0,39>
||      XOR     .S2X    B4,A5,B24       ; |159| <0,39>
||      STW     .D2T1   A3,*+B5(12)     ; |154| <0,39>
||      STW     .D1T2   B4,*+A18(12)    ; |157| <0,39>

        SHL     .S2     B9,24,B7        ; |143| <0,40>
||      SHRU    .S1     A4,16,A4        ; |148| <0,40>
||      XOR     .L2X    A5,B23,B25      ; |159| <0,40>
||      STW     .D1T2   B4,*A18         ; |142| <0,40>
        SHL     .S2     B8,8,B7         ; |151| <0,41>
||      XOR     .L1     A5,A4,A3        ; |148| <0,41>
```

```
||    XOR     .S1     A6,A3,A4         ; |144| <0,41>
||    XOR     .L2     B4,B7,B4         ; |143| <0,41>
||    STNDW   .D1T2  B25:B24,*+A18(12);|159| <0,41>

      MV      .L1     A3,A5            ; |150| <0,42>
||    STW     .D2T1   A4,*+B5(4)       ; |146| <0,42>
||    XOR     .S1X    A4,B22,A4        ; |150| <0,42>
||    STW     .D1T2   B4,*A18          ; |143| <0,42>

      XOR     .L1X    A3,B7,A3         ; |151| <0,43>
||    STNDW   .D1T1   A5:A4,*+A18(4) ; |150| <0,43>

      STW     .D1T1   A3,*+A18(8)      ; |151| <0,44>
      LDW     .D2T2   *B16,B4          ; |163| <0,45>
      NOP             4
      EXTU    .S2     B4,8,24,B8       ; |163| <0,50>
      XORMPY  .M2     B17,B8,B7        ; |181| <0,51>
      NOP             1
      XORMPY  .M1X    A19,B8,A3        ; |179| <0,53>

      LDW     .D1T1   *+A18(12),A5     ; |183| <0,54>
||    XORMPY  .M2     B20,B8,B4        ; |168| <0,54>
||    XORMPY  .M1X    A20,B8,A4        ; |187| <0,54>

      LDW     .D1T1   *+A18(4),A6      ; |172| <0,55>
||    XORMPY  .M2     B19,B8,B9        ; |170| <0,55>

      LDW     .D1T1   *A18,A17         ; |168| <0,56>
||    SHL     .S1X    B7,16,A8         ; |182| <0,56>

      LDW     .D1T1   *-A18(4),A17     ; |170| <0,57>
||    XORMPY  .M2     B18,B8,B21       ; |175| <0,57>
||    SHL     .S1     A3,24,A9         ; |181| <0,57>

      LDW     .D1T1   *+A18(16),A4     ; |183| <0,58>
||    SHRU    .S1     A4,24,A7         ; |187| <0,58>

      LDW     .D1T2   *+A18(8),B7      ; |172| <0,59>
||    SHRU    .S2     B9,24,B21        ; |172| <0,59>
||    SHRU    .S1X    B4,16,A16        ; |168| <0,59>

      XOR     .L2X    A6,B21,B7        ; |172| <0,60>
||    SHRU    .S1X    B7,16,A6         ; |183| <0,60>

      SHL     .S2     B9,8,B7          ; |171| <0,61>
||    SHL     .S1X    B4,16,A16        ; |170| <0,61>
||    XOR     .L1     A5,A6,A6         ; |183| <0,61>
||    XOR     .D1     A17,A16,A5       ; |168| <0,61>
||    XOR     .L2     B7,B21,B4        ; |175| <0,61>
||    STW     .D2T2   B7,*+B5(4)       ; |174| <0,61>

      XORMPY  .M1X    A21,B8,A4        ; |190| <0,62>
||    XOR     .L1     A17,A16,A16      ; |170| <0,62>
||    SHL     .S2X    A4,8,B8          ; |189| <0,62>
||    STW     .D1T2   B4,*+A18(4)      ; |177| <0,62>
||    STW     .D2T1   A6,*+B5(12)      ; |185| <0,62>
||    MV      .S1     A5,A17           ; |170| <0,62>

      XOR     .L1     A4,A7,A3         ; |187| <0,63>
||    SHRU    .S2X    A3,8,B9          ; |179| <0,63>
||    XOR     .S1X    A5,B7,A4         ; |171| <0,63>
||    STNDW   .D1T1   A17:A16,*-A18(4)  ; |170|
<0,63>

      MV      .L1     A3,A5            ; |189| <0,64>
||    XOR     .L2     B7,B9,B7         ; |179| <0,64>
||    XOR     .S1X    A6,B8,A4         ; |189| <0,64>
||    STW     .D1T1   A4,*A18          ; |171| <0,64>

      XOR     .L2X    B4,A9,B8         ; |181| <0,65>
||    MV      .S2     B7,B9            ; |181| <0,65>
||    STNDW   .D1T1   A5:A4,*+A18(12); |189| <0,65>

      XOR     .L2X    B7,A8,B4         ; |182| <0,66>
||    STNDW   .D1T2   B9:B8,*+A18(4) ; |181| <0,66>

      XOR     .L1     A3,A4,A3         ; |190| <0,67>
||    STW     .D1T2   B4,*+A18(8)      ; |182| <0,67>

      STW     .D1T1   A3,*+A18(16)     ; |190| <0,68>
      LDW     .D2T2   *B16++,B4        ; |194| <0,69>
      NOP             4
      SHRU    .S2     B4,24,B4         ; |194| <0,74>
      NOP             1

      XORMPY  .M2     B17,B4,B21       ; |211| <0,76>
||    XORMPY  .M1X    A21,B4,A16       ; |219| <0,76>

      XORMPY  .M2     B19,B4,B22       ; |200| <0,77>
||    XORMPY  .M1X    A20,B4,A5        ; |217| <0,77>

      LDW     .D1T2   *+A18(4),B7      ; |202| <0,78>
||    XORMPY  .M2     B18,B4,B9        ; |206| <0,78>

      LDW     .D1T1   *+A18(8),A3      ; |202| <0,79>
||    XORMPY  .M2     B20,B4,B8        ; |198| <0,79>

      LDW     .D1T1   *A18,A6          ; |198| <0,80>
||    SHL     .S1     A16,8,A9         ; |220| <0,80>

      LDW     .D1T1   *-A18(4),A7      ; |200| <0,81>
||    XORMPY  .M1X    A19,B4,A3        ; |209| <0,81>
||    SHL     .S1     A5,16,A21        ; |219| <0,81>

      LDW     .D1T1   *+A18(12),A22    ; |213| <0,82>
||    SHRU    .S2     B9,24,B4         ; |206| <0,82>
||    SHL     .S1X    B22,16,A8        ; |201| <0,82>

      LDW     .D2T1   *+B5(20),A6      ; |221| <0,83>
||    SHRU    .S2     B22,16,B22       ; |202| <0,83>
||    SHRU    .S1X    B21,8,A17        ; |213| <0,83>

      LDW     .D1T1   *+A18(16),A6     ; |213| <0,84>
||    SHL     .S2     B9,8,B9          ; |208| <0,84>
||    XOR     .L2X    A3,B4,B7         ; |206| <0,84>
||    XOR     .D2     B7,B22,B4        ; |202| <0,84>
||    SHRU    .S1X    B8,8,A4          ; |198| <0,84>

      SHL     .S1X    B8,24,A6         ; |200| <0,85>
||    XOR     .L1     A6,A4,A4         ; |198| <0,85>
||    MV      .L2     B7,B9            ; |208| <0,85>
||    XOR     .S2     B4,B9,B8         ; |208| <0,85>
||    STW     .D2T2   B4,*+B5(4)       ; |204| <0,85>

[ B0]   SUB     .L2     B0,1,B0        ; |99| <0,86>
||    SHL     .S2     B21,24,B8        ; |212| <0,86>
||    XOR     .L1     A7,A6,A6         ; |200| <0,86>
||    STNDW   .D1T2   B9:B8,*+A18(4) ; |208| <0,86>
||    MV      .S1     A4,A7            ; |200| <0,86>

      XOR     .L2X    B7,A3,B4         ; |209| <0,87>
|| [ B0]  B          .S2     $C$L2     ; |99| <0,87>
||    SHRU    .S1     A16,24,A7        ; |221| <0,87>
||    XOR     .L1     A22,A17,A3       ; |213| <0,87>
||    STNDW   .D1T1   A7:A6,*-A18(4) ; |200| <0,87>

      SHRU    .S1     A5,16,A7         ; |217| <0,88>
||    XOR     .L1     A7,A6,A5         ; |221| <0,88>
||    XOR     .L2     B4,B8,B4         ; |212| <0,88>
||    STW     .D1T2   B4,*+A18(8)      ; |211| <0,88>

      XOR     .L1     A6,A7,A4         ; |217| <0,89>
||    STW     .D2T1   A5,*+B5(20)      ; |221| <0,89>
||    XOR     .S1     A4,A8,A5         ; |201| <0,89>
||    MV      .L2X    A3,B4            ; |213| <0,89>
||    STW     .D1T2   B4,*+A18(8)      ; |212| <0,89>

      MV      .L1     A4,A7            ; |219| <0,90>
||    XOR     .S1     A3,A21,A6        ; |219| <0,90>
||    STW     .D1T1   A5,*A18          ; |201| <0,90>
||    STW     .D2T2   B4,*+B5(12)      ; |215| <0,90>

      XOR     .L1     A4,A9,A3         ; |220| <0,91>
||    STNDW   .D1T1   A7:A6,*+A18(12); |219| <0,91>

      STW     .D1T1   A3,*+A18(16)     ; |220| <0,92>
      NOP             5
```

Optimized C Code Snippet from $GF(2^{163})$ Multi-Precision Binary Polynomial Multiplication

```
225 for(i=0; i<5; i++)
226 {
227
228  b = 0xFF&opb[i];
229
230  p = _xormpy(shifted[0], b);
231  p1 = _xormpy(shifted[1], b);
232  p2 = _xormpy(shifted[2], b);
233  p3 = _xormpy(shifted[3], b);
234  p4 = _xormpy(shifted[4], b);
235  p5 = _xormpy(shifted[5], b);
236  p6 = _xormpy(shifted[6], b);
237
238  final[i] ^= p;
239  final[i] ^= (p1 << 24);
240  final[i+1] ^= (p1 >> 8);
241  final[i+1] ^= (p2 << 16);
242  final[i+2] ^= (p2 >> 16);
243  final[i+2] ^= (p3 << 8);
244  final[i+3] ^= (p3 >> 24);
245  final[i+3] ^= p4;
246  final[i+3] ^= (p5 << 24);
247  final[i+4] ^= (p5 >> 8);
248  final[i+4] ^= (p6 << 16);
249  final[i+5] ^= (p6 >> 16);
250
251
252  b = (0xFF00&opb[i]) >> 8;
253
254  p = _xormpy(shifted[0], b);
255  p1 = _xormpy(shifted[1], b);
256  p2 = _xormpy(shifted[2], b);
257  p3 = _xormpy(shifted[3], b);
258  p4 = _xormpy(shifted[4], b);
259  p5 = _xormpy(shifted[5], b);
260  p6 = _xormpy(shifted[6], b);
261
262  final[i] ^= (p << 8);
263  final[i+1] ^= (p >> 24);
264  final[i+1] ^= (p1);
265  final[i+1] ^= (p2 << 24);
266  final[i+2] ^= (p2 >> 8);
267  final[i+2] ^= (p3 << 16);
268  final[i+3] ^= (p3 >> 16);
269  final[i+3] ^= (p4 << 8);
270  final[i+4] ^= (p4 >> 24);
271  final[i+4] ^= (p5);
272  final[i+4] ^= (p6 << 24);
273  final[i+5] ^= (p6 >> 8);
274
275
276
277  b = (0xFF0000&opb[i]) >> 16;
278
279
280  p = _xormpy(shifted[0], b);
281  p1 = _xormpy(shifted[1], b);
282  p2 = _xormpy(shifted[2], b);
283  p3 = _xormpy(shifted[3], b);
284  p4 = _xormpy(shifted[4], b);
285  p5 = _xormpy(shifted[5], b);
286  p6 = _xormpy(shifted[6], b);
287
288  final[i] ^= (p << 16);
289  final[i+1] ^= (p >> 16);
290  final[i+1] ^= (p1 << 8);
291  final[i+2] ^= (p1 >> 24);
292  final[i+2] ^= (p2);
293  final[i+2] ^= (p3 << 24);
294  final[i+3] ^= (p3 >> 8);
295  final[i+3] ^= (p4 << 16);
296  final[i+4] ^= (p4 >> 16);
297  final[i+4] ^= (p5 << 8);
298  final[i+5] ^= (p5 >> 24);
299  final[i+5] ^= p6;
300
301
302
303  b = (0xFF000000&opb[i])>>24;
304
305  p = _xormpy(shifted[0], b);
306  p1 = _xormpy(shifted[1], b);
307  p2 = _xormpy(shifted[2], b);
308  p3 = _xormpy(shifted[3], b);
309  p4 = _xormpy(shifted[4], b);
310  p5 = _xormpy(shifted[5], b);
311  p6 = _xormpy(shifted[6], b);
312
313  final[i] ^= (p << 24);
314  final[i+1] ^= (p >> 8);
315  final[i+1] ^= (p1 << 16);
316  final[i+2] ^= (p1 >> 16);
317  final[i+2] ^= (p2 << 8);
318  final[i+3] ^= (p2 >> 24);
319  final[i+3] ^= (p3);
320  final[i+3] ^= (p4 << 24);
321  final[i+4] ^= (p4 >> 8);
322  final[i+4] ^= (p5 << 16);
323  final[i+5] ^= (p5 >> 16);
324  final[i+5] ^= (p6 << 8);
325  final[i+6] ^= (p6 >> 24);
326
327
328 }
```

Assembly Output from Compiling Optimized C Code Snippet from $GF(2^{163})$ Multi-Precision Binary Polynomial Multiplication

```
;*----------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 99
;*      Loop opening brace source line   : 100
;*      Loop closing brace source line   : 202
;*      Known Minimum Trip Count         : 5
;*      Known Maximum Trip Count         : 5
;*      Known Max Trip Count Factor      : 5
;*      Loop Carried Dependency Bound(^) : 32
;*      Unpartitioned Resource Bound     : 24
;*      Partitioned Resource Bound(*)    : 32
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                    29       18
;*      .D units                     8       32*
;*      .M units                    20        8
;*      .X cross paths              13       28
;*      .T address paths            20       32*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0
;*      Addition ops (.LSD)         23       27
;*      Bound(.L .S .LS)            15        9
;*      Bound(.L .S .D .LS .LSD)    20       26
;*
;*----------------------------------------------------------*
$C$L3:; PIPED LOOP PROLOG
;** --------------------------------------------------------*
        XORMPY  .M1     A8,A7,A6      ; |110| <0,7>

        LDW     .D1T1   *++A9,A5      ; |114| <0,8>
||      EXTU    .S1     A22,8,24,A21  ; |155| <0,8>
||      XORMPY  .M1     A18,A7,A4     ; |107| <0,8>

        LDW     .D1T1   *+A9(16),A27  ; |123| <0,9>
||      EXTU    .S1     A22,16,24,A24 ; |130| <0,9>

        LDW     .D2T2   *B8++,B6      ; |114| <0,10>
||      LDW     .D1T1   *+A9(12),A23  ; |123| <0,10>
||      XORMPY  .M1     A19,A7,A6     ; |106| <0,10>
||      XORMPY  .M2X    B21,A7,B16    ; |114| <0,10>
||      SHL     .S1     A4,24,A3      ; |118| <0,10>

        LDW     .D2T2   *+B8(8),B5    ; |118| <0,11>
||      XORMPY  .M2X    B20,A24,B4    ; |132| <0,11>
||      SHRU    .S1     A4,8,A23      ; |123| <0,11>

        LDW     .D2T2   *+B8(4),B16   ; |118| <0,12>
||      XORMPY  .M2X    B20,A7,B9     ; |108| <0,12>
||      SHL     .S1     A6,16,A25     ; |123| <0,12>

        SHRU    .S1     A6,16,A26     ; |123| <0,13>
||      SHRU    .S2X    A4,24,B4      ; |118| <0,13>

        XOR     .L1     A25,A23,A4    ; |123| <0,14>
||      SHL     .S1     A4,8,A3       ; |118| <0,14>
||      SHRU    .S2     B16,8,B7      ; |114| <0,14>
||      XOR     .L2X    A3,B4,B18     ; |118| <0,14>

        XORMPY  .M1     A20,A7,A4     ; |114| <0,15>
||      SHL     .S1     A6,16,A4      ; |114| <0,15>
||      XOR     .L1     A27,A26,A7    ; |123| <0,15>
||      SHRU    .S2X    A6,16,B17     ; |118| <0,15>
||      XOR     .D1     A23,A4,A6     ; |123| <0,15>

        XORMPY  .M1     A8,A24,A3     ; |134| <0,16>
||      SHL     .S1X    B16,24,A3     ; |114| <0,16>
||      XOR     .L2X    A3,B17,B5     ; |118| <0,16>
||      XOR     .S2     B5,B18,B17    ; |118| <0,16>
||      STNDW   .D1T1   A7:A6,*+A9(12); |123| <0,16>

        XORMPY  .M2X    B20,A21,B16   ; |158| <0,17>
||      XOR     .L1X    A4,B7,A6      ; |114| <0,17>
||      XOR     .L2     B16,B5,B16    ; |118| <0,17>
||      LDW     .D2T2   *+B8(12),B6   ; |147| <0,17>
||      XOR     .S2     B9,B17,B17    ; |118| <0,17>

        XORMPY  .M1     A20,A24,A4    ; |137| <0,18>
||      XOR     .L1X    B6,A3,A3      ; |114| <0,18>
||      STNDW   .D2T2   B17:B16,*+B8(4); |118| <0,18>

        XORMPY  .M1     A19,A24,A5    ; |130| <0,19>
||      XOR     .L1     A5,A6,A5      ; |114| <0,19>
||      XOR     .S1     A4,A3,A4      ; |114| <0,19>
||      LDW     .D2T1   *+B8(16),A7   ; |147| <0,19>

        XORMPY  .M1     A17,A24,A3    ; |133| <0,20>
||      SHL     .S1     A3,24,A6      ; |147| <0,20>
||      STNDW   .D1T1   A5:A4,*-A9(4) ; |114| <0,20>

        XORMPY  .M2X    B21,A21,B9    ; |155| <0,21>
||      SHRU    .S2     B16,16,B17    ; |172| <0,21>
||      SHRU    .S1X    B4,24,A5      ; |147| <0,21>
||      LDW     .D2T2   *B8,B7        ; |137| <0,21>

        XORMPY  .M2X    B21,A24,B6    ; |137| <0,22>
||      SHL     .S2     B4,8,B5       ; |142| <0,22>
||      XOR     .L1     A6,A5,A6      ; |147| <0,22>
||      LDW     .D2T2   *-B8(4),B18   ; |137| <0,22>

        XORMPY  .M1     A18,A24,A3    ; |131| <0,23>
||      SHRU    .S1     A3,8,A6       ; |147| <0,23>
||      SHRU    .S2X    A4,24,B4      ; |137| <0,23>
||      LDW     .D2T2   *+B8(4),B18   ; |142| <0,23>
||      XOR     .L1X    B6,A6,A23     ; |147| <0,23>

        SHL     .S2X    A5,24,B6      ; |137| <0,24>
||      LDW     .D2T2   *+B8(8),B6    ; |142| <0,24>
||      XOR     .L1     A7,A6,A7      ; |147| <0,24>
||      XOR     .S1     A3,A23,A6     ; |147| <0,24>

        XORMPY  .M1     A20,A21,A4    ; |163| <0,25>
||      SHRU    .S2X    A5,8,B19      ; |142| <0,25>
||      XOR     .L2     B6,B4,B4      ; |137| <0,25>
||      STNDW   .D2T1   A7:A6,*+B8(12); |147| <0,25>

        XORMPY  .M1     A17,A21,A3    ; |159| <0,26>
||      SHL     .S2X    A4,8,B7       ; |137| <0,26>
||      XOR     .L2     B7,B4,B4      ; |137| <0,26>
||      LDW     .D2T2   *+B8(12),B4   ; |172| <0,26>

        XORMPY  .M1     A18,A21,A5    ; |157| <0,27>
||      SHL     .S1     A3,16,A5      ; |142| <0,27>
||      XOR     .L2     B18,B7,B6     ; |137| <0,27>
||      LDW     .D2T2   *+B8(16),B5   ; |172| <0,27>
||      XOR     .S2     B6,B4,B7      ; |137| <0,27>

        SHRU    .S1     A22,24,A23    ; |180| <0,28>
||      SHRU    .S2X    A3,16,B4      ; |142| <0,28>
||      STNDW   .D2T2   B7:B6,*-B8(4) ; |137| <0,28>

        SHRU    .S1     A4,16,A7      ; |163| <0,29>
||      XOR     .L2     B5,B4,B5      ; |142| <0,29>
||      XOR     .S2X    A5,B19,B4     ; |142| <0,29>
||      LDW     .D2T2   *B8,B5        ; |163| <0,29>

        XORMPY  .M1     A8,A21,A3     ; |160| <0,30>
||      SHL     .S1     A3,8,A6       ; |172| <0,30>
||      XOR     .L2     B6,B5,B7      ; |142| <0,30>
||      XOR     .S2     B18,B4,B6     ; |142| <0,30>
||      LDW     .D2T2   *-B8(4),B17   ; |163| <0,30>

        XORMPY  .M1     A20,A23,A4    ; |188| <0,31>
||      XORMPY  .M2X    B21,A23,B6    ; |180| <0,31>
```

```
||   SHL     .S2     B16,16,B7      ; |168| <0,31>
||   SHL     .S1X    B9,8,A22       ; |163| <0,31>
||   STNDW   .D2T2   B7:B6,*+B8(4)  ; |142| <0,31>

     XORMPY  .M1     A19,A21,A3     ; |156| <0,32>
||   SHRU    .S1     A3,24,A3       ; |172| <0,32>
||   SHL     .S2X    A5,24,B16      ; |168| <0,32>
||   XOR     .L1X    A6,B17,A6      ; |172| <0,32>
||   LDW     .D2T2   *+B8(4),B5     ; |168| <0,32>

     XORMPY  .M1     A17,A23,A6     ; |184| <0,33>
||   XORMPY  .M2X    B20,A23,B9     ; |183| <0,33>
||   SHRU    .S2     B9,24,B9       ; |168| <0,33>
||   XOR     .L1     A22,A7,A21     ; |163| <0,33>
||   LDW     .D2T2   *+B8(8),B7     ; |168| <0,33>
||   XOR     .S1X    B5,A3,A7       ; |172| <0,33>

     XORMPY  .M1     A19,A23,A5     ; |181| <0,34>
||   SHL     .S2X    A4,16,B4       ; |163| <0,34>
||   XOR     .L1X    B4,A6,A6       ; |172| <0,34>
||   XOR     .S1     A3,A7,A7       ; |172| <0,34>

     STNDW   .D2T1   A7:A6,*+B8(12); |172| <0,35>
||   XOR     .L2     B17,B4,B4      ; |163| <0,35>
||   XOR     .S2X    B5,A21,B5      ; |163| <0,35>

     XORMPY  .M1     A8,A23,A6      ; |185| <0,36>
||   SHRU    .S1     A4,8,A21       ; |188| <0,36>
||   SHRU    .S2X    A5,8,B16       ; |168| <0,36>
||   XOR     .L2     B16,B9,B4      ; |168| <0,36>
||   STNDW   .D2T2   B5:B4,*-B8(4)  ; |163| <0,36>

     SHL     .S1     A6,16,A25      ; |197| <0,37>
||   XOR     .L2     B7,B16,B5      ; |168| <0,37>
||   XOR     .S2     B5,B4,B4       ; |168| <0,37>
||   LDW     .D2T2   *+B8(12),B4    ; |197| <0,37>

     SHL     .S1     A5,8,A24       ; |192| <0,38>
||   XOR     .L2     B7,B5,B5       ; |168| <0,38>
||   XOR     .S2X    A3,B4,B4       ; |168| <0,38>
||   LDW     .D2T2   *+B8(16),B7    ; |197| <0,38>

     SHRU    .S1     A6,16,A7       ; |197| <0,39>
||   STNDW   .D2T2   B5:B4,*+B8(4)  ; |168| <0,39>

     SHL     .S1     A6,8,A8        ; |197| <0,40>
```

```
||   LDW     .D2T2   *+B8(8),B6     ; |192| <0,40>

     LDW     .D1T1   *+A9(20),A3    ; |199| <0,41>
||   SHL     .S1X    B6,16,A3       ; |188| <0,41>
||   LDW     .D2T2   *-B8(4),B7     ; |188| <0,41>

[A0] SUB     .L1     A0,1,A0        ; |99| <0,42>
||   SHRU    .S1X    B9,8,A22       ; |197| <0,42>
||   LDW     .D2T2   *B8,B9         ; |188| <0,42>

     XOR     .L1     A8,A7,A4       ; |197| <0,43>
||   SHRU    .S1     A5,24,A8       ; |192| <0,43>
||   XORMPY  .M1     A18,A23,A4     ; |182| <0,43>
||   SHL     .S2X    A4,24,B5       ; |188| <0,43>
||   LDW     .D2T2   *+B8(4),B6     ; |192| <0,43>
||   LDW     .D1T1   *A16++,A22     ; |102| <1,0>

     XOR     .L1     A25,A22,A4     ; |197| <0,44>
||   XOR     .D1X    B7,A4,A5       ; |197| <0,44>
||[A0] B     .S1     $C$L4          ; |99| <0,44>
||   SHRU    .S2     B6,16,B7       ; |192| <0,44>

     SHRU    .S1     A6,24,A6       ; |199| <0,45>
||   SHL     .S2     B9,24,B4       ; |192| <0,45>
||   XOR     .L2X    A24,B7,B16     ; |192| <0,45>
||   XOR     .L1     A3,A21,A7      ; |188| <0,45>
||   XOR     .D1X    B4,A4,A4       ; |197| <0,45>
||   LDW     .D2T1   *+SP(32),A8    ; |110| <1,2>

     XOR     .L1     A6,A3,A3       ; |199| <0,46>
||   XOR     .L2X    B4,A8,B4       ; |192| <0,46>
||   STNDW   .D2T1   A5:A4,*+B8(12); |197| <0,46>

     STW     .D1T1   A3,*+A9(20)    ; |199| <0,47>
||   XOR     .L2     B6,B4,B7       ; |192| <0,47>
||   XOR     .S2     B7,B5,B4       ; |188| <0,47>
||   XOR     .D2X    B9,A7,B5       ; |188| <0,47>

     XOR     .L2     B6,B16,B4      ; |192| <0,48>
||   STNDW   .D2T2   B5:B4,*-B8(4)  ; |188| <0,48>
||   XOR     .S2X    A4,B7,B5       ; |192| <0,48>
||   EXTU    .S1     A22,24,24,A7   ; |106| <1,5>

     STNDW   .D2T2   B5:B4,*+B8(4)  ; |192| <0,49>
||   XORMPY  .M1     A17,A7,A4      ; |109| <1,6>
```

CHAPTER 5

RESULTS

In this chapter, we discuss how we compiled our implementation. We compare our implementation to a standard library and other implementations on an ARM, and we analyze our results.

5.1 Tools and Environment

We wrote our implementations in C and was able to execute the XORMPY instruction by using the "_xormpy()" function call. We compiled these implementations using the TI C6x C Compiler using the -O3 option. It was run on the C64x+ at 360MHz on an OMAP3530.

We compared our results by running modular multiplication using libcrypt on the ARM Cortex A8 on the same OMAP3530. libcrypt is the cryptographic library which OpenSSL uses. We compiled libcrypt using GCC with the -O3 option.

5.2 Execution Times on the OMAP3530

Table III shows our results. We can perform finite field multiplication faster using the DSP despite the DSP running at a slower clock rate. Using our implementation, we can perform binary field multiplication over seven times faster on the DSP when compared to using libcrypt on the ARM for $GF(2^{283})$ and smaller. We can perform $GF(2^{233})$ multiplications on the DSP over six times faster than prime field $GF(p224)$ multiplications on the faster ARM. There is a drop off in speed up as the field increases in size, with the largest drop off at $GF(2^{571})$. Figure 8 is a graph of the execution times of binary field multiplication on the OMAP3530. These comparisons are relevant because these are two

TABLE III. MODULAR MULTIPLICATION ON THE TI OMAP3530

NIST Field	500MHz ARM Cortex A8 libcrypt		360MHz TI C64x+ DSP Our Implementation		
	Time (µs)	Cycles	Time (µs)	Cycles	Speed Up
$GF(2^{163})$	6.62	3310	0.829	298	7.99
$GF(2^{233})$	9.98	4990	1.31	472	7.62
$GF(2^{283})$	15.44	7720	2.20	792	7.02
$GF(2^{409})$	27.47	13735	4.43	1595	6.20
$GF(2^{571})$	45.53	22765	11.60	4175	3.93
$GF(p192)$[a]	6.80[b]	3400[b]	-	-	-
$GF(p224)$[a]	8.79[b]	4395[b]	-	-	-
$GF(p256)$[a]	9.28[b]	4640[b]	-	-	-

a. Montgomery Multiplication
b. Does not include overhead needed for Montgomery Multiplication

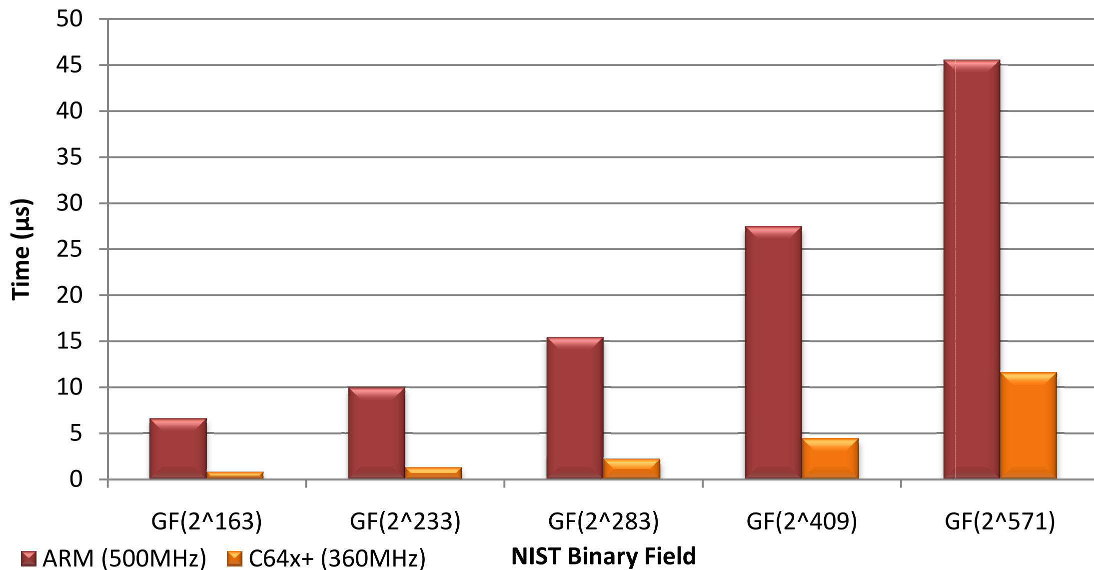

Figure 8. Execution times of multiplications in all NIST Binary Fields on the ARM Cortex A8 using libcrypt and C64x+ using our implementation.

processors implemented on the same SoC. So our results have factored out differences in technology.

Figure 9 depicts a graph of multiplication execution cycles per bit vs. bits in the NIST Field. It shows that as the number of bits in the NIST Field increase, the cycles per bit needed to complete the multiplication also increases. Figure 10 shows the cycles per bit

vs. bits^2 in the NIST Field. This is close to a horizontal line. So the multiplication in our implementation is $O(n^2)$, which is a result of us multiplying each byte of the one operand by each subword of the other operand. Figure 11 shows that the libcrypt implementation

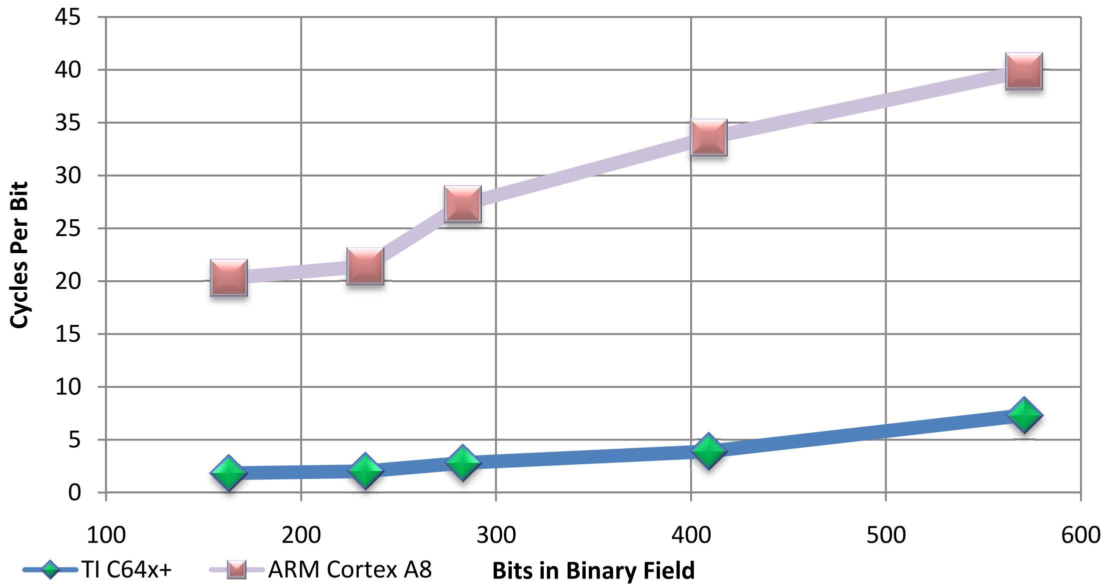

Figure 9. Multiplication execution cycles per bit versus the number of bits in the NIST Binary Field on the ARM Cortex A8 (libcrypt) and TI C64x+ (our implementation).

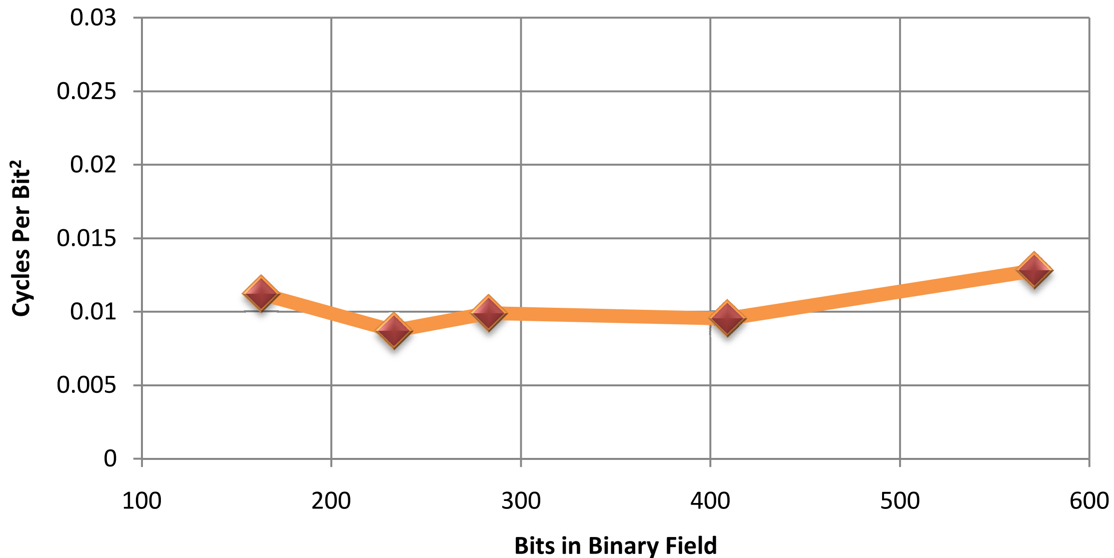

Figure 10. Multiplication execution cycles per bit^2 versus the number of bits in the NIST Binary Field using our implementation on the TI C64x+.

running on the ARM Cortex A8 decreases in cycles per bit^2 in a NIST Binary Field. This shows that the libcrypt implementation uses a form of Karatsuba Multiplication, which is O($n^{\log_2 3}$). Figure 12 shows the number of cycles for the libcrypt multiplication vs. the

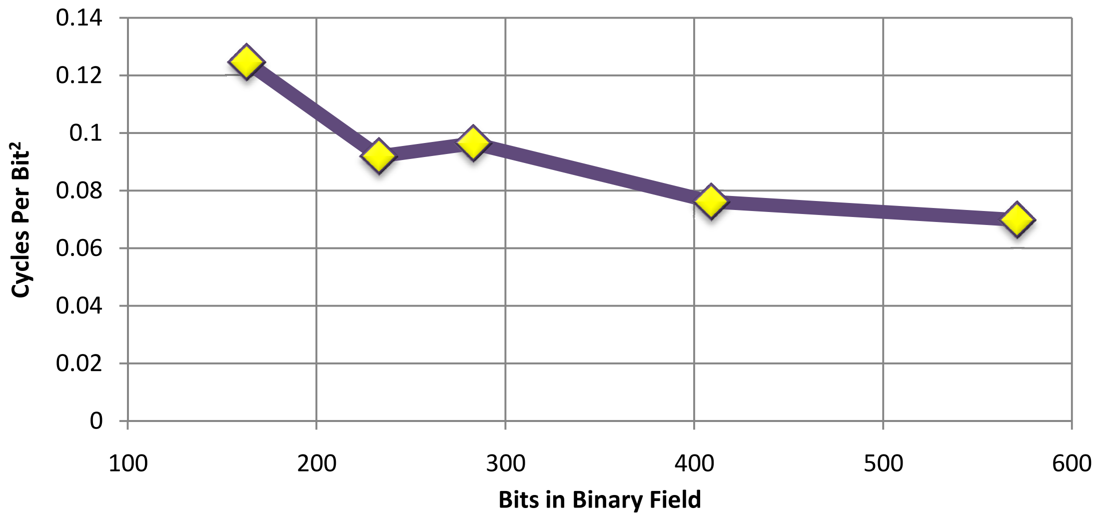

Figure 11. Multiplication execution cycles per bit^2 versus the number of bits in the NIST Binary Field using our implementation on the ARM Cortex A8.

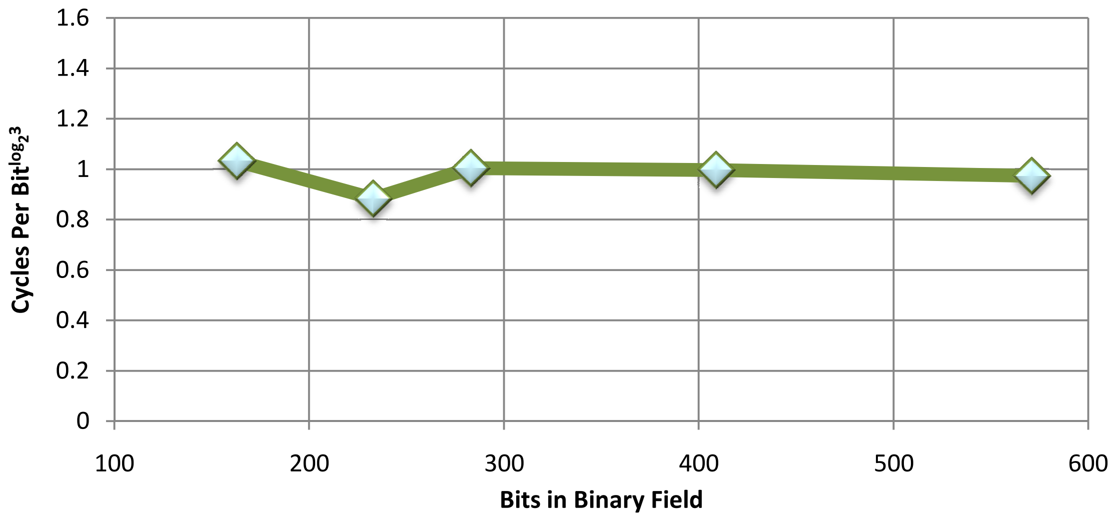

Figure 12. Multiplication Execution Cycles per bit raised to the $\log_2 3$ power versus the number of bits in the NIST Binary Field using libcrypt on the ARM Cortex A8.

number of bits raised to the power of $\log_2 3$. This displays a horizontal line, and shows that libcrypt does use a form of Karatsuba Multiplication. This gives us the reason why our speed up decreases as the number of bits in the field increase, i.e. the libcrypt implementation uses Karatsuba Multiplication, while our implementation does not.

5.3 Cycle Count Compared To Other Implementations

Table IV shows our results compared to results shown in other papers. In [14], an ARM instruction extension, MULGF, was implemented. MULGF performs the same operations as XORMPY, except that it takes two 32-bit operands and produces a 63-bit result. Figure 13 shows the execution cycles of our implementation compared with the implementations in [14] with the MULGF instruction implemented. Even though the MULGF is more powerful than XORMPY, i.e. it can produce a 63-bit result as opposed to a 32-bit result, the C64x+ still outperforms the ARM by big margin. This shows that because of the highly parallelized architecture of the C64x+, it can perform modular multiplication much faster than the ARM Cortex A8.

TABLE IV. CYCLES FOR MODULAR MULTIPLICATION

Processor	**Implementation**	**NIST Field**	**Cycles**
TI C64x+ DSP	Our Implementation	$GF(2^{163})$	298
		$GF(2^{233})$	472
		$GF(2^{283})$	792
ARM	Montgomery [15]	$GF(p256)$	3384
	Pencil and Paper [14]	$GF(2^{163})$	~8000
	Montgomery [14]	$GF(2^{163})$	~24000
	Karatsuba [14]	$GF(2^{163})$	~5500
ARM + MULGF instruction extension	Pencil and Paper [14]	$GF(2^{163})$	~2400
	Montgomery [14]	$GF(2^{163})$	~7000
	Karatsuba [14]	$GF(2^{163})$	~2500

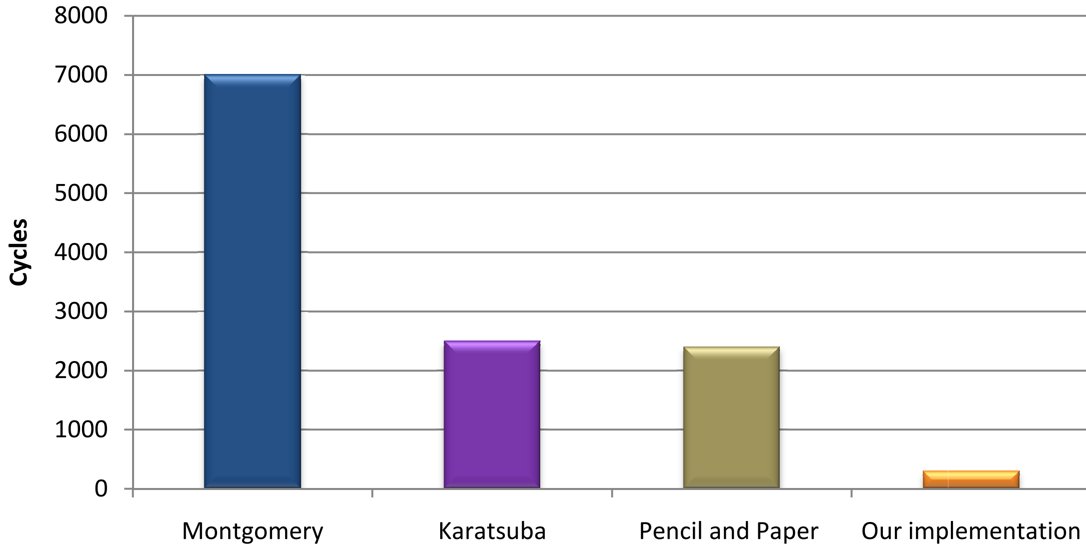

Figure 13. Execution cycles of multiplication implementations on an ARM processor with the MULGF instruction [14] and our implementation on the C64x+ DSP in $GF(2^{163})$.

5.4 Reduction Results

We implemented both of our reduction techniques for $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$ and $GF(2^{409})$. Our results are listed in Table V. Using the Shift and Add method works better only for $GF(2^{233})$ and $GF(2^{409})$. This supports our claim that if most of the set bits in the irreducible polynomial are within a 9-bit window, then it is more effective to use Multi-Precision Binary Polynomial Multiplication. Figure 14 displays this data in a graph.

TABLE V. CYCLES FOR UNBALANCED EXPONENT MODULAR REDUCTION

NIST Field	**Irreducible Polynomial**	**Cycles for Reduction Method**	
		Shift and Add	**Multi-Precision Multiplication**
$GF(2^{163})$	$x^{163}+x^7+x^6+x^3+x^0$	28	19
$GF(2^{233})$	$x^{233}+x^{74}+x^0$	33	42
$GF(2^{283})$	$x^{283}+x^{12}+x^7+x^5+x^0$	90	64
$GF(2^{409})$	$x^{409}+x^{87}+x^0$	52	66

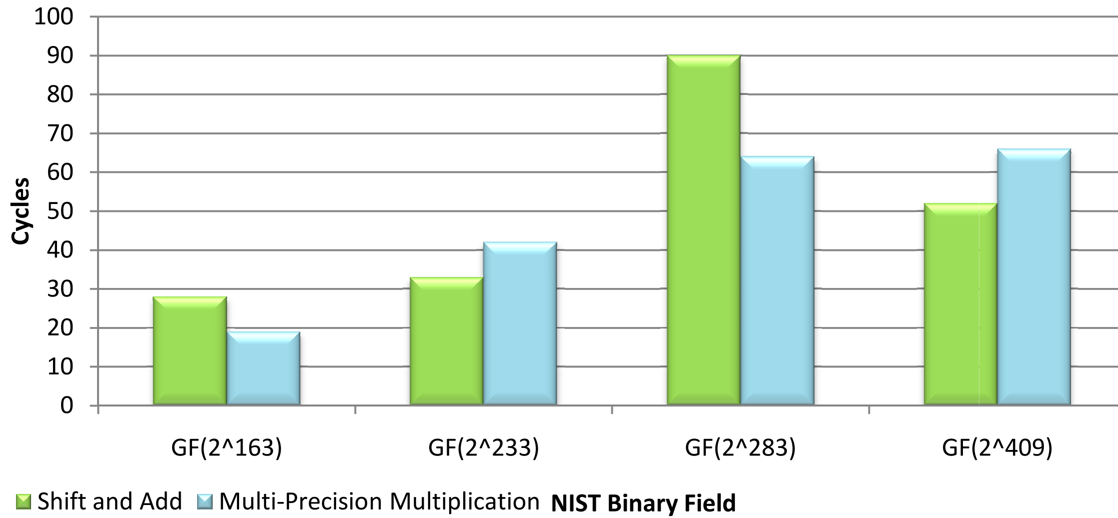

Figure 14. Cycles for Unbalanced Exponent Modular Reduction using the shift and add method and MPBPM.

One might expect that reduction for $GF(2^{409})$ would take longer than reduction for $GF(2^{283})$ because it is a larger binary field. However, $T(x)$ for $GF(2^{283})$ has more bits set in it when compared with the $T(x)$ for $GF(2^{409})$, and not all bits are within a 9 bit window. Therefore, the first iteration of UEMR requires the "reduction byte", (0xA1), to be multiplied by $C_h(x)$, and then $C_h(x)$ must be shifted and added to that result. Since in $GF(2^{283})$ there are 282 bits in $C_h(x)$, there are $\lceil 282/24 \rceil = 12$ subwords that need to be multiplied by the reduction byte in the first iteration. Then $C_h(x)$ must be shifted and added to account for the bit not in the reduction byte, i.e. x^{12}. So there are $\lceil 282/32 \rceil = 9$ adds and 9 2 = 18 shifts needed in addition to the adds and shifts needed for binary polynomial multiplication. In the first iteration of reduction for $GF(2^{409})$, there needs to be only $\lceil 408/32 \rceil = 13$ adds and 13 2 = 26 shifts.

Table VI displays the percentage of cycles of multiplications needed for Unbalanced Exponent Modular Reduction. The reduction method which needed the least number of cycles was chosen for each binary field. Figure 15 shows this data in pie graphs. This data shows that reduction for $GF(2^{283})$ takes the highest percentage of execution time when compared with the other NIST Binary Fields. This is because it

must perform both binary polynomial multiplication using XOR-Multiply instructions and shift and add. Figure 15 shows that reduction does count for a very small percentage of the total multiplication time, which supports our usage of UEMR. Since the number of bits set in $T(x)$ are either 2 or 4, as the NIST Binary Field becomes larger, the percentage of execution time used for reduction becomes smaller.

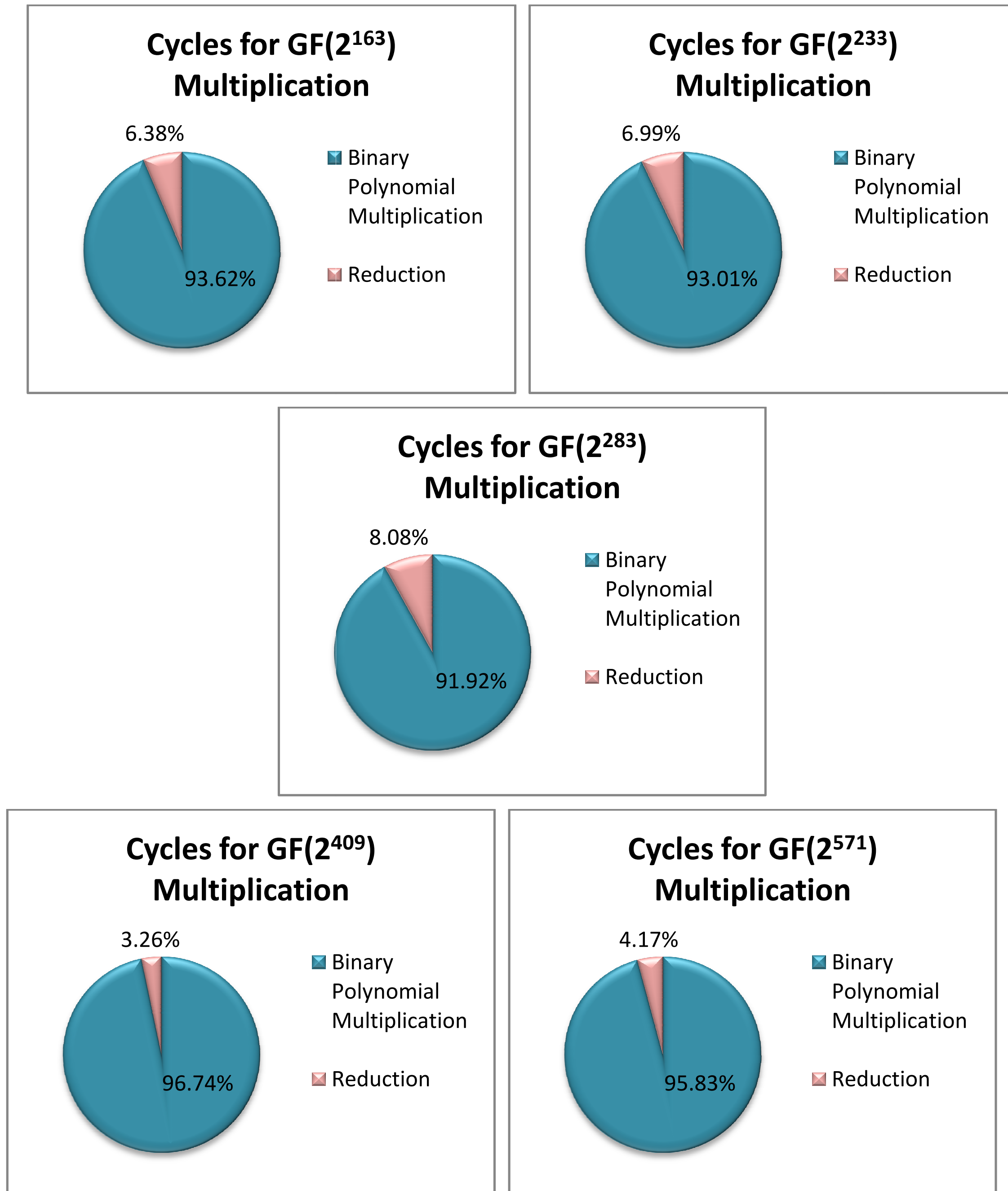

Figure 15. Percentage of execution time needed for reduction for each NIST Binary Field in our implementation on the C64x+

TABLE VI. CYCLE PERCENTAGE FOR REDUCTION

NIST Field	**Irreducible Polynomial**	**Percentage of cycles for MPBPM**	**Percentage of cycles for Reduction**
$GF(2^{163})$	$x^{163} + x^7 + x^6 + x^3 + x^0$	93.62%	6.38%
$GF(2^{233})$	$x^{233} + x^{74} + x^0$	93.01%	6.99%
$GF(2^{283})$	$x^{283} + x^{12} + x^7 + x^5 + x^0$	91.92%	8.08%
$GF(2^{409})$	$x^{409} + x^{87} + x^0$	96.74%	3.26%
$GF(2^{571})$	$x^{571} + x^{10} + x^5 + x^2 + x^0$	95.83%	4.17%

CHAPTER 6

FUTURE WORK

6.1 Improving Binary Field Multiplication

While our results were very good, there is room for improvement. Our lowest speed ups came at larger binary fields. Our algorithm can be enhanced by incorporating Karatsuba Multiplication in it so it can perform in less than $O(n^2)$ time. The larger binary fields would benefit most from this.

To represent numbers in $GF(2^{571})$ on the C64x+, ⌈571/32⌉ = 18 registers are needed. Let x_0 and y_0 represent the least significant 9 registers of x and y, respectively. Let x_1 and y_1 represent the most significant 9 registers of x and y, respectively. Let the base b be 2^{288} ($2^{(9)(32)}$). Instead of calculating $(x_1y_1)b^2 + (x_0y_1 + x_1y_0)b + x_0y_0$, you can use Karatsuba Multiplication and calculate $(x_1y_1)b^2 + x_0y_0 + ((x_1 + x_0)(y_0+ y_1) - x_0y_0 - x_1y_1)b$. This increases the number of additions and subtractions, equivalient to Exclusive-ORs, but significantly decreases the number of XOR-Multiplies needed. Six XOR instructions can be executed every cycle and these instructions have no delay slots. Only two XOR-Multiply instructions can be dispatched each cycle and require three delay slots. Using XOR-Multiply also requires executing XOR and shift instructions to create partial products. Using Karatsuba Multiplication in this instance decreases the number of XOR-Multiply instructions needed for MPBPM from 1728 to 1296, while increasing the number 32-bit XOR instructions by 18. This is a significant drop off in the the number of instructions needed.

We implemented this on the C64x+ and it produced very good results. The execution time decreased from 11.60μs to 6.81μs. The speed up compared to the libcrypt implementation on the ARM increased from 3.93 to 6.68. Karatsuba Multiplication can also be performed recursively. That is, each multiplication required, i.e. x_1y_1, x_0y_0 and $(x_1 + x_0)(y_0+ y_1)$, in this new implementation can be also be performed using Karatsuba

Multiplication. This requires each multiplication to be broken up into smaller parts to decrease the total number of XOR-Multiply instructions. This can potentially be done several times to further increase performance. There will be a point where performing another Karatsuba decomposition will make the execution time longer. Therefore further work can be done to further optimize these algorithms for this DSP.

Functions to perform binary field squaring can be also be developed based on MPBPM and UEMR. To square x, you get $x^2 = (x_1x_1)b^2 + (x_0x_1 + x_0x_1)b + x_0x_0$. Since adding terms in a binary field is equivalent to performing a bit-wise XOR operation, the middle term is equal to zero because $x_0x_1 + x_0x_1 = 0$. Hence $x^2 = (x_1x_1)b^2 + x_0y_0$. Therefore squaring a binary field polynomial can be performed much faster than even Karatsuba Multiplication.

6.2 Implementing a Cryptosystem on the C64x+

Modular multiplication is just one part of a modern cryptosystems. Implementing an entire cryptosystem on the C64x+ can further support our argument that utilizing a VLIW DSP for cryptographic operations can significantly reduce execution time while freeing up the General Purpose Processor. Even though we have showed that the most computationally intensive part of an encryption algorithm can be sped up greatly, implementing a whole cryptographic system might show bottlenecks that might slow down the system. Cache and memory access might limit the C64x+. Additionally, if a full cryptosystem was implemented, there would need to be communication with the ARM or other peripherals. So communication may prove to be a bottleneck in this application.

6.3 Finite Field Multiplications on Other Processors

Future research can be done to see how finite field multiplication can perform on other processors. One can survey newer DSPs, newer ARM processors or other low power processors. Power analysis of different processors performing modular multiplication would also be interesting.

CHAPTER 7

CONCLUSIONS

We have shown that migrating applications from an ARM to a DSP can provide drastic improvements in performance. We implemented binary field multiplication using Multi-Precision Binary Polynomial Multiplication on the TI C64x+. For most fields, this provided a six times speed up when compared with the ARM on the same chip.

We have shown that Unbalanced Exponent Modular Reduction can also be efficiently implemented on the C64x+ using both the shift and add method and Multi-Precision Binary Polynomial Multiplication.

We have explained how our algorithm can be improved by taking advantage of Karatsuba Multiplication. We also explored other possible future work.

We have taken our source code and produced a C library for the C64x+ which can perform binary field addition and multiplication for all NIST Binary Fields. Our source code is posted online and is available for all to view, edit and compile. It is downloadable from: http://rijndael.ece.vt.edu/ctergino

REFERENCES

[1] "BeagleBoard System Reference Manual Rev C2," beagleboard.org, revision 0.2, March 2009.

[2] D. Hankerson, A. Menezes and S. Vanstone, Guide to Elliptic Curve Cryptography. New York, NY: Springer, 2004, pp. 213-215.

[3] S. Haibin, Y. Jin, and R. You, "Unbalanced Exponent Modular Reduction over Binary Field and Its Implementation," Innovative Computing, Information and Control, 2006. ICICIC '06. First International Conference on, vol. 1, pp. 190-193, Aug-Sept 2006.

[4] M. Knezevic, K. Sakiyama, J. Fan, and I. Verbauwhede, "Modular Reduction in $GF(2^n)$ Without Pre-Computational Phase," Lecture Notes in Computer Science, Arithmetic of Finite Fields - 2nd International Workshop, WAIFI 2008, Proceedings, pp. 77-87, July 2008.

[5] J. Sankaran, "Reed Solomon Decoder: TMS320C64x Implementation," Digital Signal Processing Solutions Application Report, Texas Instruments, December 2000.

[6] P. Montgomery, "Multiplication Without Trial Division," Mathematics of Computation, vol. 44, issue 170, pp. 519-521, April 1985.

[7] C. Koc, T. Acar and B. Kaliski, "Analyzing and Comparing Montgomery Multiplication Algorithms," IEEE Micro, vol. 16, issue 16, pp. 26-33, June 1996.

[8] B. Sunar, "A generalized method for constructing subquadratic complexity GF(2k) multipliers," IEEE Transactions on Computers, vol. 53, issue 9, pp. 1097-1105, Sept. 2004.

[9] "OMAP3530/25 Applications Processor," Texas Instruments, May 2009.

[10] "Architecture and Implementation of the ARM® Cortex™-A8 Microprocessor," ARM, October 2005.

[11] "TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide," Texas Instruments, October 2008.

[12] K. Itoh, M. Takenaka, N. Torii, S. Temma and Y. Kurihara, " Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201," Lecture Notes In Computer Science, vol 1717, pp. 61-72, 1999.

[13] P. Gastaldo, G. Parodi and R. Zunino, "Enhanced Montgomery Multiplication on DSP Architectures for Embedded Public-Key Cryptosystems," EURASIP Journal on Embedded Systems, vol. 8, issue 3, pp. 1-9, April 2008.

[14] S. Bartolini, I. Branovic, R. Giogri, and E. Martinelli, "Effects of Instruction-Set Extensions on an Embedded Processor: A Case Study on Elliptic Curve Cryptography over $GF(2^m)$," IEEE Transactions on Computers, vol. 57, issue 5, pp. 672-686, May 2008.

[15] A. Tenca and C. Koc, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm," IEEE Transactions on Computers, vol. 52, issue 9, pp. 1215-1221, Sept 2003.