Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

4-1-2008

Implementing efficient 384-Bit NIST elliptic curves over prime fields on an ARM946E

Tracy VanAmeron

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
VanAmeron, Tracy, "Implementing efficient 384-Bit NIST elliptic curves over prime fields on an ARM946E" (2008). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Implementing Efficient 384-Bit NIST Elliptic Curve over Prime Fields on an ARM946E

by

Tracy VanAmeron

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Professor Mike Eastman
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
April, 2008

Approved By:

Professor Mike Eastman
Primary Advisor – R.I.T. Dept. of Computer Engineering Technology

---

Professor Marcin Lukowiak
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Professor Dhireesha Kudithipudi
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Mike Kurdziel
Secondary Advisor – Harris RF Communications

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Implementing Efficient 384-Bit NIST Elliptic Curves over Prime Fields on an ARM946E

I, TRACY VANAMERON, HEREBY GRANT PERMISSION TO THE WALLACE MEMORIAL LIBRARY TO REPRODUCE MY THESIS IN WHOLE OR PART.

____
Tracy VanAmeron

____
Date

Dedication

The goal of completing this thesis was not reached alone. I had the encouragement and support of so many people along the way. I have been blessed with family, friends, and colleagues who have been very supportive. I dedicate my Thesis to them.

My Husband and Children

To my husband Jeremy, who supported and encouraged me to reach this goal. I love you and don't know what I would ever do without you.

To my boys, Zachery and Justin, you are my pride and joy. And to my unborn daughter Alexis, I can't wait to hold you for the first time. I love each of you so very much.

My Family

To my aunt, Laurie Pask, you have always wanted me to finish this thesis and achieve my master's degree. You provided encouragement for me to complete a task that at times seemed never ending. You have shown me more strength than anyone I have ever known, and wanted to make sure I completed this as much for you as did for myself.

To my grandmother, Grace Heschke, I love you and miss you more than words can express. I think of you often and wish you were here to see just how far I have come, to meet your great grandchildren, and for them to meet you. You were so strong, and taught me so much.

// Acknowledgements

This work was completed with the help of a few people that I would like to acknowledge and thank.

I would like to thank my advisor, **Professor Mike Eastman**, who taught me so much along my way. You provided support and encouragement not only as my advisor of this thesis, but as a teacher and mentor in my undergraduate studies.

I would like to thank my committee members, **Professor Marcin Lukowiak** and **Professor Dhireesha Kudithipudi**, for their suggestions and contributions.

To my supervisor and committee member, **Dr. Mike Kurdziel** of Harris Corporation, thank you for the support, encouragement, and guidance you provided throughout the duration of this project.

I would also like to thank my colleagues at Harris Corporation who provided their support, encouragement, and ideas during my research.

Abstract

This thesis presents a performance evaluation of a 384-bit NIST elliptic curve over prime fields on a 32-bit ARM946E microprocessor running at 100-MHz. While adhering to the constraints of an embedded system, the following items were investigated to decrease computation time: the importance of the underlying finite arithmetic, the use of hardware accelerators, the use of memory options, and the use of available processor features.

The elliptic curve implementation utilized existing finite arithmetic C code to interface to an AiMEC Montgomery Exponentiator Core. The exponentiator core supports modular addition, modular multiplication, and exponentiation. The finite arithmetic C code also contained functions to perform operations which are not performed by the exponentiator such as non-modular multiplication, non-modular addition, and modular subtraction.

Multiple enhancements were made to the finite field arithmetic. These provided a 22% time reduction in execution time of the 384-bit elliptic curve multiplication. Enhancements included writing assembly functions, adding checks prior to performing a modular reduction, utilizing the exponentiator core only when modulus reduction was necessary, using multiplication if more than two additions are required and placing the finite arithmetic into its own library and using ARM mode. Other optimizations investigated including: cache usage, compiler options (speed vs. size), and Thumb instruction set vs. ARM instruction set provided minimal reduction, 3.6%, in the execution time.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 ELLIPTIC CURVE CRYPTOGRAPHY **2**

CHAPTER 3 ELLIPTIC CURVES **4**

3.1. ELLIPTIC CURVE GROUPS 4

3.1.1 Groups over Real Numbers 5

3.1.2 Groups over Binary Fields 6

3.1.3 Groups over Prime Fields 6

3.2. ELLIPTIC CURVE POINT REPRESENTATION 6

3.3. ELLIPTIC CURVE OPERATIONS 8

3.3.1 Point Addition 8

3.3.2 Point Doubling 11

3.3.3 Elliptic Curve Point Multiplication 14

CHAPTER 4 THE ARM946E PROCESSOR **18**

4.1. THE ARM946E EMBEDMENT 20

4.1.1 The AiMEC Montgomery Exponentiator Core 22

4.1.1.1 Modular Exponentiation 22

4.1.1.2 Modular Multiplication and Addition 23

4.1.1.3 AiMEC Montgomery Exponentiator Core Summery 24

CHAPTER 5 FINITE ARITHMETIC DESIGN AND TEST SETUP **25**

5.1. FINITE ARITHMETIC WRITTEN FOR THE ARM946E 25

5.2. TEST SETUP 27

CHAPTER 6 EVALUATION AND TIMING DESCRIPTIONS **28**

6.1. BASELINE TIMING DESCRIPTION 28

6.2. HARDWARE VS. SOFTWARE OPTIMIZATIONS 29

6.3. FINITE ARITHMETIC SOFTWARE OPTIMIZATIONS 30

6.3.1 Finite Arithmetic Library Compiled in ARM Mode 32

6.4. COMPILER OPTIMIZATION SETTINGS 33

6.5. SIDE CHANNEL ATTACKS REMOVED 34

6.6. MEMORY OPTIONS 35

CHAPTER 7 SUMMARY AND FUTURE WORK 37

BIBLIOGRAPHY 41

APPENDIX A 43

A.1 FINITE ARITHMETIC ASSEMBLY ROUTINES 43

A.2 ELLIPTIC CURVE C ROUTINES 48

List of Figures

Figure 1: Key Sizes 2

Figure 2: Elliptic Curve Equation $y^2 = x^3 - 3x + 1$ [1] 5

Figure 3: Point Addition [1] 9

Figure 4: Adding P and –P [1] 10

Figure 5: Point Doubling [1] 12

Figure 6: Point Doubling when Py = 0 [1] 13

Figure 7: ARM946E Embedment with Peripherals 21

Figure 8: Test Setup Diagram 27

Figure 9: Elliptic Curve Multiply Baseline and Performance Optimization 1 30

Figure 10: Elliptic Curve Multiply with Performance Optimizations 1 and 2 31

Figure 11: Elliptic Curve Multiply with Performance Optimizations 2 and 3 33

Figure 12: Elliptic Curve Multiply with Performance Optimizations 3 and 4 34

Figure 13: Other Code Options 36

Figure 14: Finite Arithmetic and ARM Optimization Used 38

List of Algorithms

Algorithm 1: Point Addition using Jacobian and Affine Coordinates 11

Algorithm 2: Point Doubling using Jacobian Coordinates 13

Algorithm 3: Point Multiplication – binary method 14

Algorithm 4: Point Multiplication – Lem-Lee and Shamir's... 16

Algorithm 5: Assembly Routine mult_32 26

List of Tables

Table 1: Overview of all options tested 39

Glossary

AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
CRAM	Configurable Random Access Memory
EBIU	External Bus Interface Unit
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECCs	Elliptic Curve Crypto Systems
ECDH	Elliptic-Curve Diffie-Hellman
ECDSA	Elliptic-Curve Digital Signature Algorithm
ECMQV	Elliptic-Curve Menezes-Qu-Vanstone
FPGA	Field Programmable Gate Array
ICE	Integrated Circuit Emulator
NIST	National Institute of Standards and Technology
NSA	National Security Agency
PC	Personal Computer
RISC	Reduced Instruction Set Computer
SRAM	Static Random Access Memory
TCM	Tightly-Coupled Memory

Chapter 1 Introduction

Due to the need for increased security, cryptography has become a necessity in many embedded applications. It has become an integral portion of embedded applications that include hand held radios used by law enforcement, hand held computers, smart cards, and other industries wanting the ability to maintain and transfer sensitive data. Elliptic Curve Cryptosystems (ECCs) have quickly become the cryptography of choice because of the reduced overhead required in comparison to other cryptosystems to achieve the same level of security, i.e. 160-bit Elliptic Curve Crypto Systems provide the same security as 1024-bit RSA[i].

Elliptic curve algorithms, used in Elliptic Curve Crypto Systems, are a series of hundreds to thousands of finite arithmetic operations. A great deal of research has been performed to determine the most efficient algorithms to implement elliptic curve arithmetic, particularly the use of large processors that are very efficient at performing multi-precision arithmetic. There has been limited research focused on placing this complex arithmetic into an embedded system with small, low power processors with, potentially limited, multi-precision arithmetic capability.

Chapter 2 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) was proposed as an alternative mechanism for implementing public-key cryptography in 1985 by Victor Miller (IBM) and Neil Koblitz (University of Washington) [1]. Public-key algorithms are a method for sharing keys among two or more participants to form a secure channel. Unlike other popular algorithms such as RSA[i], Elliptic Curve Cryptography is based on discrete logarithms that are more computationally expensive to invert at equivalent key lengths [1]. As can be seen by NIST (National Institute of Standards and Technology) guidelines, in Figure 1, reduced overhead (number of transmitted bits of data) is required for Elliptic Curve Cryptography in comparison to RSA to achieve the same level of security. Initially, even though Elliptic Curve Cryptography offered greater security per key bit, its complex elliptic curve algorithm made Elliptic Curve Cryptography computationally intensive and thus it was not widely used. Since initial implementations, there has been a large amount of research focused on discovering more efficient algorithms to improve the overall performance of Elliptic Curve Cryptography.

NIST guidelines for public key sizes for AES

ECC KEY SIZE (Bits)	RSA KEY SIZE (Bits)	KEY SIZE RATIO	AES KEY SIZE (Bits)
163	1024	1 : 6	
256	3072	1 : 12	128
384	7680	1 : 20	192
512	15 360	1 : 30	256

Supplied by NIST to ANSI X9F1

Figure 1: Key Sizes

Elliptic Curve Cryptography has been included in commercial standards such as ANSI (American National Standards Institute), IEEE (Institute of Electrical and Electronics Engineers), and NIST (National Institute of Standards and Technology) [2]. Beyond the commercial standards, there are currently three National Security Agency (NSA) Suite B public key algorithms that are based on Elliptic Curve arithmetic methods.

These public key algorithms are; Elliptic-Curve Menezes-Qu-Vanstone (ECMQV), Elliptic-Curve Diffie-Hellman (ECDH), and Elliptic-Curve Digital Signature Algorithm (ECDSA).

Chapter 3 Elliptic Curves

This section first discusses elliptic curve groups and how they are formed, followed by elliptic curve point representations. Next is a discussion on the state of the art in elliptic curve research. This is followed by an introduction to the graphical representation and the algorithm chosen for the elliptic curve addition, elliptic curve double, and elliptic curve multiply functionality.

3.1. Elliptic Curve Groups

Elliptic Curve Cryptosystems require the use of abstract algebraic groups. Together with elliptic curve geometry, these arithmetic set definitions are used to form elliptic curve groups. A group is a set of elements that are closed under a mathematical operation called the Group Function or Group Operation. For elliptic curve groups, the Group Operation is defined geometrically. By definition, the Elliptic Curve Group Operation is chosen to be associative and discrete. The number of member elements is constrained to a finite number of curve points while still remaining closed under the Group Function. Lastly, each curve contains an inverse element and an identity element.

Elliptic curve groups can be generated with the underlying fields of F_p (*where p is a prime*) and F_2m (*a binary representation with 2^m elements*) [1]. An elliptic curve must satisfy the equation $y^2 = x^3 + ax + b$, where a and b are constants satisfying $4a^3 + 27b^2 \neq 0$. Each curve also contains a special point O, called the point of infinity.

3.1.1 Groups over Real Numbers

An elliptic curve over real numbers can be described by a set of points (x,y) satisfying the equation $y^2 = x^3 + ax + b$ where *x, y, a*, and *b* are real numbers. Each choice of a and b will generate a different elliptic curve. For example, the curve that has a = –3 and b = 1 is shown in Figure 2. If $x^3 + ax + b$ contains no repeated factors, or equivalently if $4a^3 + 27b^2$ is not equal to 0, then the elliptic curve $y^2 = x^3 + ax + b$ can be used to form a group, along with the point of infinity, O [1]. Calculations using real numbers are slow and inaccurate due to rounding errors. Elliptic curve cryptosystems require fast and accurate arithmetic; therefore, real numbers are not used in elliptic curve cryptosystems.

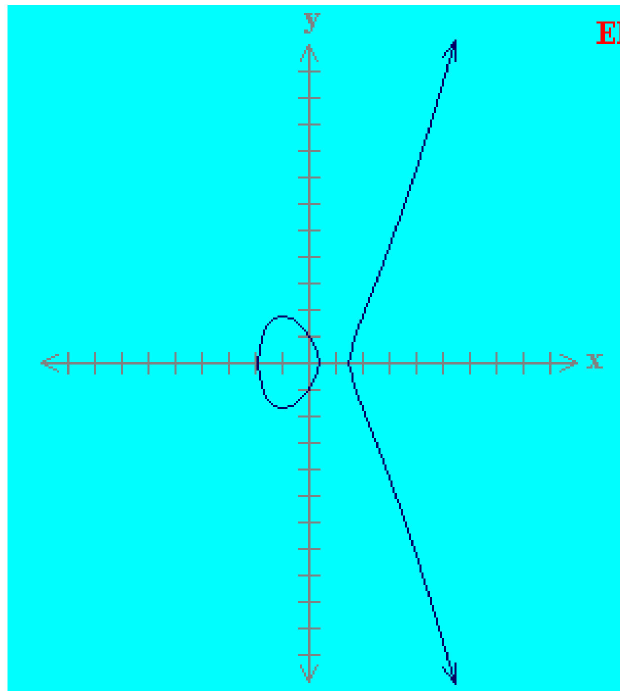

Figure 2: Elliptic Curve Equation $y^2 = x^3 - 3x + 1$ [1]

3.1.2 Groups over Binary Fields

The elements of binary fields, F_2m, are represented using a polynomial basis representation with reduction polynomial f(x) [2], thus the elements are m-bit strings. An elliptic curve over F_2m is specified by the coefficients a and b which are elements of F_2m of the defining equation $y^2 + xy = x^3 + ax^2 + b$. The elliptic curve includes all points (x,y) which satisfy the elliptic curve equation over F_2m (where x and y are elements of F_2m). An elliptic curve group over F_2m consists of the points on the corresponding elliptic curve, together with a point at infinity, O [1].

3.1.3 Groups over Prime Fields

The elements of prime fields, $F_{p,}$ use the numbers from 0 to p – 1, and all calculations are completed by performing a modulo reduction by p. An Elliptic Curve E over prime fields, $F_{p,}$ is the algebraic curve described by a set of points (x,y) satisfying the equation $y^2 \; mod \; p = (x^3 + ax + b) \; mod \; p$ where a, b are elements of $F_{p.}$ The number of points on the elliptic curve over F_p is nh, where n is prime and h is called the cofactor.

NIST recommends these elliptic curves over prime fields: F_{192}, F_{224}, F_{256}, F_{384}, and F_{521}. The 384-bit NIST Elliptic Curve over the prime field F_p where

$p = 2^{384} – 2^{128} – 2^{96} + 2^{32} – 1,\ a = –3,\ h = 1,\ n$ and b are also provided. The focus of this thesis will be on the NIST curves over prime fields.

3.2. *Elliptic Curve Point Representation*

A point on an elliptic curve may be represented by multiple point coordinate systems. The coordinate systems, which were taken from [2] are Affine where a point is

represented as (x_A, y_A). Projective points are represented as (X, Y, Z) which can be converted back to the Affine coordinate system using the equations $x_A = XZ^{-1}$ and $y_A = YZ^{-1}$. Jacobian coordinates are represented as (X, Y, Z), Modified Jacobian coordinates are represented as (X, Y, Z, aZ^4), and Chudnovsky Jacobian coordinates are represented as (X, Y, Z, Z^2, Z^3). These three coordinate systems can be converted back to the Affine coordinate system using the equations as $x_A = XZ^{-2}$ and $y_A = YZ^{-3}$.

Affine coordinates are used for communication between any two parties because they require the lowest bandwidth of all the coordinate systems. Thus, all calculations performed in other coordinate systems must be converted back to Affine coordinates for the communication process. However, the Affine coordinate system by itself is highly inefficient to perform the elliptic curve multiplication and the elliptic curve doubling because inversions would be required. Inversions are computationally more expensive than multiplication or addition.

The Affine Coordinates can be efficiently changed to any of these other coordinate systems by setting each of the Z's $(Z, Z^2$ and $Z^3)$ to 1, and aZ^4 to a (the elliptic curve parameter). Converting any of the other coordinate systems back into Affine Coordinates is computationally expensive due to the necessary inversions ($x_A = XZ^{-1}$ and $y_A = YZ^{-1}$, and $x_A = XZ^{-2}$ and $y_A = YZ^{-3}$).

All NIST Elliptic Curves are defined by the elliptic curve parameter $a = -3$. The choice of $a = -3$ for the NIST curves was made because it produces a faster algorithm for Elliptic Curve (EC) point doubling using Jacobian coordinates [2].

Since the research defined in this paper focuses on the NIST Elliptic Curves a combination of Jacobian and Affine coordinate systems will be used for the EC double and EC addition functions.

3.3. Elliptic Curve Operations

The actual algorithms described in Algorithm 1 through Algorithm 4 are based on the research described in section 3.3, and thus are those used in this implementation. The actual C-code written to perform the elliptic curve operations can be found in the Appendix, A.2, which follows the Finite Arithmetic Assembly Routines.

3.3.1 Point Addition

Graphically, the addition of two points (P + Q = R) on a curve is a simple concept. To start, simply draw a line between the two points, P and Q. This line will intersect the curve at exactly 1 more point, –R. Reflect –R across the x-axis and this new point is R. See Figure 3 for an example. It should be noted that every point, P on the curve contains a reflection, -P across the x-axis. Thus, elliptic curves are reflective across the x-axis.

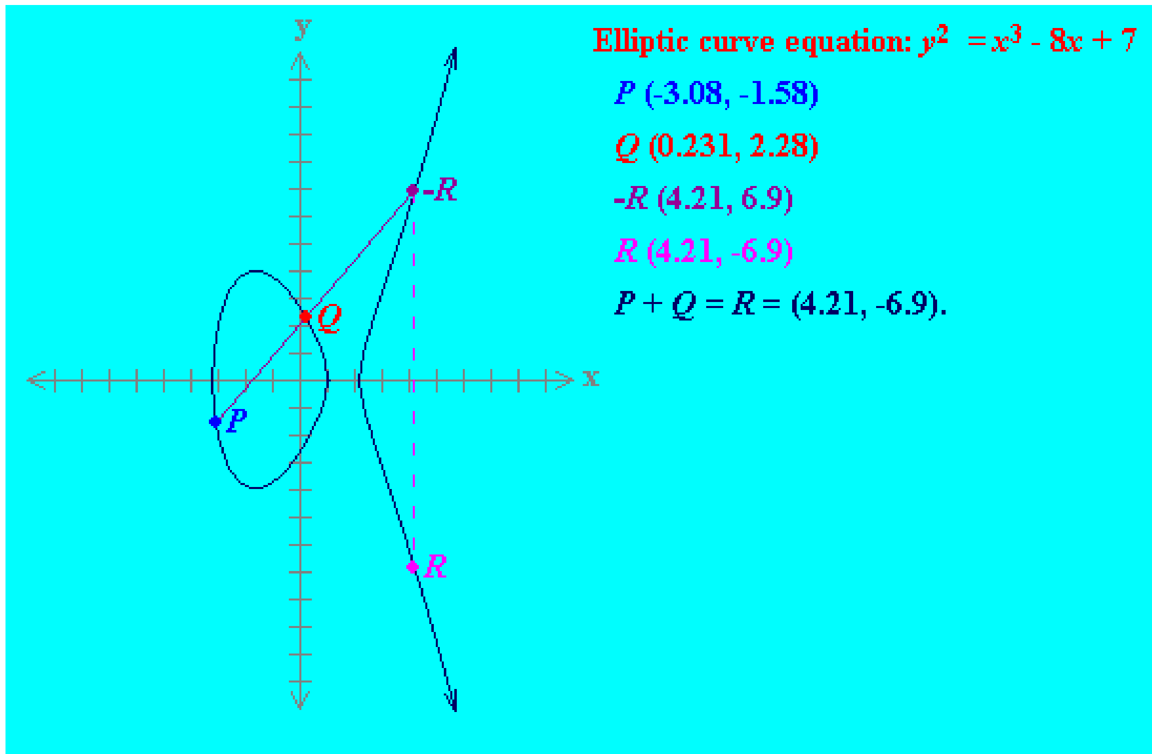

Figure 3: Point Addition [1]

Adding the points P and –P is a special case. When a line is drawn between P and –P it is vertical, and therefore never again intersects the elliptic curve. By definition, P + (-P) = O, the point of infinity. As a result of this, P + O = P, thus making O the additive identity of the elliptic curve group.

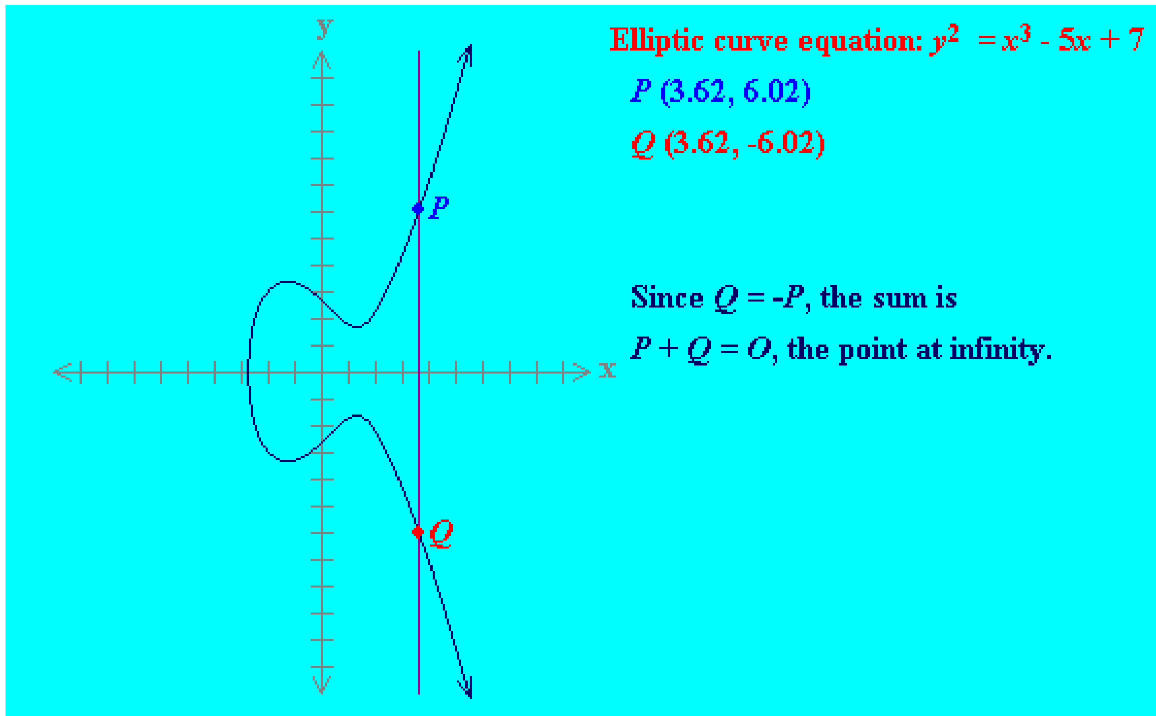

Figure 4: Adding P and –P [1]

According to [2], [3], and [4] the most efficient software algorithm for adding points on a NIST defined elliptic curve over F_p is performed using a combination of Jacobian and Affine Coordinates. However, [4] provides additional steps if both coordinates are in Jacobian format. If the 2nd parameter of [4] is modified from an Affine coordinate to a Jacobian coordinate, making $z = 1$, the algorithms are equivalent because when $z = 1$, steps are eliminated. Based on [2], [3], and [4], the elliptic curve point addition function, P(x, y, z) + Q(x, y) = R(x, y, z) for this thesis was written in software as shown in Algorithm 1.

Algorithm 1: Point Addition using Jacobian and Affine Coordinates P(x, y, z) + Q(x, y) = R(x, y, z)

$A = Q_x + P_z^2$

$B = Q_y + P_z^3$

$C = A - P_x$

$D = B - P_y$

$R_x = D^2 - (C^3 + 2 P_x * C^2)$

$R_y = D * (P_x * C^2 - Rx) - P_y * C^3$

$R_z = P_z * C$

3.3.2 Point Doubling

Similarly to graphical point addition, graphically doubling a point P + P = 2P = R is also a simple process. At point P, a tangent line is drawn. As long a P_y is not 0, the tangent line will intersect the elliptic curve at exactly one other point, –R. Reflecting –R across the x-axis yields the new point of R. Refer to Figure 5 for an example. It should be noted that 3P is found by adding P + 2P, 4P = P + 3P, etc.

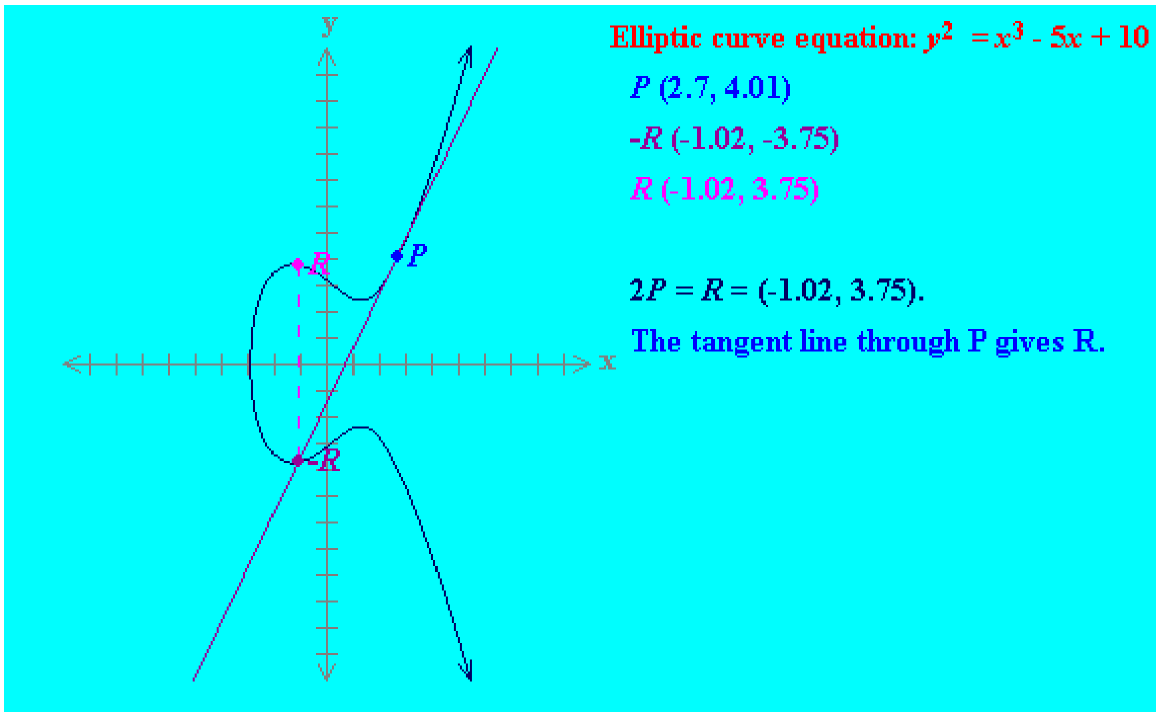

Figure 5: Point Doubling [1]

If P_y is equal to 0, then the resulting tangent line will be vertical, and will therefore never intersect the elliptic curve. In this case 2P = O, the point of infinity, and 3P becomes P + O, thus 3P = P, 4P = O, 5P = P, etc.

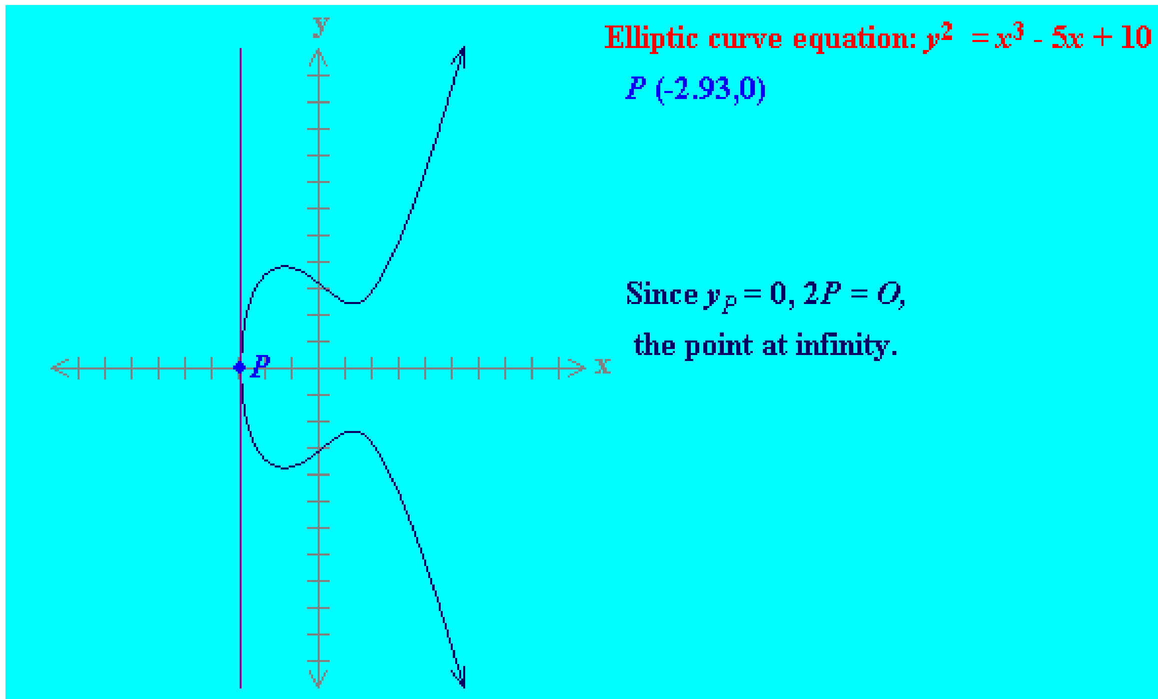

Figure 6: Point Doubling when Py = 0 [1]

According to [2], [3], & [4] the most efficient software algorithm for doubling a point on a NIST defined elliptic curve over F_p is performed using Jacobian Coordinates. The resulting elliptic curve double routine, 2P(x, y, z) = R(x, y, z), was written in software as follows:

Algorithm 2: Point Doubling using Jacobian Coordinates, 2P(x, y, z) = R(x, y, z)

$A = 4P_x + P_y^2$

$B = 8P_y^4$

$C = 3(P_x - P_z^2) * (P_x + P_z^2)$

$R_x = -2A + C^2$

$R_y = C * (A - R_x) - B$

$R_z = 2P_y * P_z$

3.3.3 Elliptic Curve Point Multiplication

Point multiplication is defined as kP, where k is a scalar, or integer, and P is an elliptic curve point. Point multiplication is the basis of Elliptic Curve Cryptography and consists of a series of point additions and point doubles. These operations are where a vast majority of computation time is spent. Algorithm 3 shows the basic method for point multiplication using a random component, R = kP.

Algorithm 3: Point Multiplication – binary method R = kP

```
R gets P
For all bits(i) of k
  R gets 2R (Point Double)
  If k(i) = 1
    Then R gets R ++ P (Point Addition)
End For
Return R
```

Algorithm 3 is one elliptic curve multiply which, for the 384-bit NIST curve, requires 383 elliptic curve doubles and on average, 191 elliptic curve additions.

By adapting the Lem-Lee technique [8] the algorithm can be rewritten to be two smaller elliptic curve multiplies by:

- Breaking k into two pieces k_1 and k_2 such that k_1 and k_2 when concatenated together form the bits of k, that is $k = k_1 \mid k_2$.
- Precalculating 2^{192} x P, that is double P 192 times.
- The computation of R can then be written as:
  - $R = k_1$ ** (2^{192} x P) ++ k_2 ** P
    - Where ++ represents an elliptic curve add and ** represents an elliptic curve multiply

These multiplies can be combined using the method of Shamir as modified by [9]. In the equation above, 2^{192} x P and P are two points on the elliptic curve and k_1 and k_2 are two scalar values. This algorithm works by precomputing the elliptic point, 2^{192} x P, and computing k_1 ** (2^{192} x P) ++ k_2 ** P all at the same time using the double and add process. This process is illustrated as shown in Algorithm 4.

If P is constant, and only k changes, then the values (2^{192} x P) and ((2^{192} x P) ++ P) can be pre-calculated and stored in memory in Affine format. Since (2^{192} x P) is P doubled 192 times, the savings is much more profound if P is constant since the number of elliptic curve double necessary is decreased by 50%. If P is not constant, these two values may be calculated at the beginning of the function and then used in the loop. However the number of savings is greatly diminished since the number of doubles remains the same as the basic elliptic curve multiply routine. For most key exchange algorithms which make use of the elliptic curves, P is constant once the curve being used is determined.

Algorithm 4: Point Multiplication – Lem-Lee and Shamir's R = k_1 ** (2^{192} x P) ++ k_2 * P

Read (2^{192} x P) and ((2^{192} x P) ++ P) from memory

If the MSB of k_1 = 0 & MSB of k_2 = 0

Then repeat this step with next MSB of k_1 & MSB of k_2

Else if MSB of k_1 = 0 & MSB of k_2 = 1

Then Set R = P

Else if MSB of k_1 = 1 & MSB of k_2 = 0

Then Set R = (2^{192} x P)

Else – MSB of k_1 = 1 & MSB of k_2 = 1

Then Set R = ((2^{192} x P) ++ P)

For all bits(i) of k_1 & all bits(i) k_2

R gets 2R (Point Double)

If $k_1(i)$ = 0 & $k_2(i)$ = 0

Do nothing

Else if $k_1(i)$ = 0 & $k_2(i)$ = 1

Then Set R = R ++ P (Point Addition)

Else if $k_1(i)$ = 1 & $k_2(i)$ = 0

Then Set R = R ++ (2^{192} x P) (Point Addition)

Else – $k_1(i)$ = 1 & $k_2(i)$ = 1

Then Set R = R ++ ((2^{192} x P) ++ P) (Point Addition)

End if

End For

Return R

Algorithm 4 requires 191 elliptic curve doubles, and on average, 144 elliptic curve additions for the 384-bit NIST curve. Using this process reduces the number of doubles by ½, and the number of adds by ¼ as compared to the basic elliptic curve R = kP algorithm depicted in Algorithm 3. Another important point is that each of the values added to R (P, (2^{192} x P), and [(2^{192} x P) ++ P]) can not only be precalculated, but may also be stored in Affine format, working nicely with the point addition which is faster if one parameter is an Affine coordinate.

The process presented using the Lem-Lee [8] technique required the pre-calculation of (2^{192} x P) and ((2^{192} x P) ++ P). Since this value is the same for all random component calculations, it can be computed once and stored for later use. To reduce computation times further, additional values can be pre-computed and stored. To use this process, the random number k is broken into 4 parts k_1, k_2, k_3, & k_4. This would require 16 pre-computed values to be stored in memory. Each of the pre-computed values would be stored in memory as an Affine point of the size of the curve being used. Thus, using the 384 bit curve, 32 384 bit numbers would need to be stored (16 x and 16 y components). Using this method could reduce computation time by almost 50%, but requires additional memory storage which may not be available in an embedded system.

Chapter 4 The ARM946E Processor

The ARM946E is a 32-bit RISC processor core for embedded applications. It is a product offered by ARM Limited. This processor is optimized for embedded applications which require high performance, low cost, small die size, and low power consumption. The processor uses a 5 stage pipeline: fetch, decode, execute, buffer/data, and write-back. The 5-stage pipeline achieves 1.1 MIPS/MHz, and approaches a single instruction per clock cycle.

The ARM946E processor core contains the enhanced DSP capability of the ARM9E-S processor core. The ARM9E-S processor core executes the extended ARM5vT instruction set which includes multiplier and saturating arithmetic functions. Multiply instructions are processed faster using a single-cycle 32x16 implementation. There are both 32x16 and 16x16 multiply instructions, and the pipeline allows one multiply to begin each cycle.

The ARM946E supports an external Tightly-Coupled Memory (TCM) for both instruction and data that can range in size from 0KB to 1MB. The TCM is made up of SRAM and the minimum size, when it exists, is 4KB incrementing in powers of 2 (ex. 8KB, 16KB) up to 1MB. TCM size is defined once embedment occurs. This memory is capable of returning data to the ARM9E-S core in a single cycle (no wait states).

Also implemented within the ARM946E are two cache memories, one for instruction and one for data. Like the TCM, the caches are also made up of SRAM and can range in size from 0KB to 1MB. The sizes of the data and instruction cache can be selected independently. If cache exists, the minimum size is 4KB incrementing in

powers of 2 up to 1MB. The caches connect to the ARM9E-S core through 32-bit buses and allow the pre-fetch of one instruction each cycle. This allows the Load And Store Multiple instructions to transfer one register per cycle. Cache lock-down is provided to allow critical code to be locked into cache, which ensures predictability for the code. The cache replacement algorithm is programmable and may be selected to be either pseudo random or round-robin. Cache entries are added on a read miss basis. The write policy is also programmable and may be write-back or write-through.

The ARM9E-S processor core contains 31 general purpose registers, 16 of which are visible to the user at any one time. The remaining registers are used to speed up exception processing. Register 15 is used as the program counter, R14 contains the return address following a subroutine call, and R13 is used as the stack pointer.

Another important feature of the ARM946E is that it contains two instruction sets. It implements both the traditional 32-bit wide ARM instruction set and the 16-bit Thumb instruction set. The Thumb instruction set encodes a subset of the 32-bit ARM instructions, and removes the limitations of code density and performance from narrow memory. Thumb instructions are basically compressed ARM instructions which are decompressed at execution time, and then executed as ARM instructions on the processor. The decompression is simple and performed on the fly without additional cycles. Thumb instructions have higher performance than ARM instructions on a processor with a 16-bit data bus, but lower performance on a 32-bit data bus. Thumb mode is ideal for memory constrained systems; on average the same C code is about 30% smaller if compiled using the Thumb instruction set rather than the ARM instruction set.

The processor is capable of switching between ARM and Thumb mode, although extra code, called veneer, is sometimes necessary to carry out the transition.

4.1. The ARM946E Embedment

The block diagram for the ARM946E embedment utilized for this thesis is shown in Figure 7. Internal memory available includes 16 Kbytes of TCM for instruction use and 16 Kbytes of TCM for data use. The Cache includes 16 Kbytes and is set up as 4 - way set associative using pseudo random cache replacement and write-back. Cache may be enabled for CRAM or external SRAM, but not TCM since accesses to the TCM already contain no wait states.

On the AMBA bus is 32 Kbytes of CRAM (Configurable Random Access Memory). Interfacing to the EBIU (External Bus Interface Unit), which is also connected to the AMBA, are flash and more SRAM. Additionally, within the ARM processor are 16 Kbytes of data cache and 16 Kbytes of instruction cache. The ARM946E embedment utilizes an AiMEC Montgomery Exponentiator Core which is attached to the peripheral bus controller and interfaces to the AMBA bus. Both the processor and the exponentiator core run at 100 MHz.

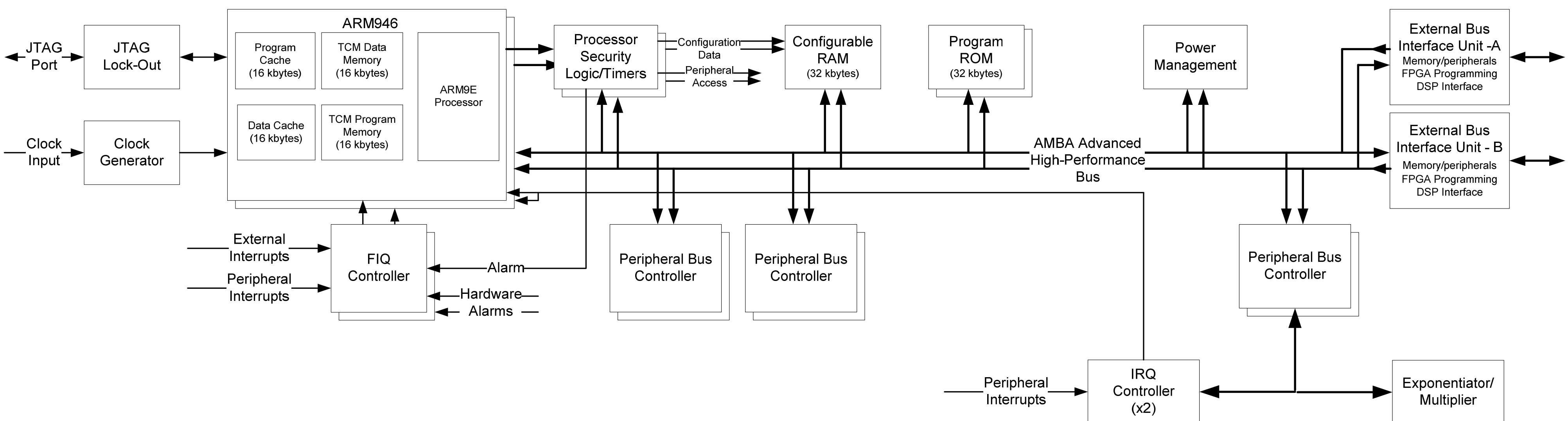

Figure 7: ARM946E Embedment with Peripherals
Image provided by Harris Corporation

4.1.1 The AiMEC Montgomery Exponentiator Core

The AiMEC Montgomery Exponentiator Core used in this implementation is capable of three independent modes of operation:

1. Montgomery Exponentiation: $x = a^e \bmod n$
2. Modular Multiplication: $m = (p * q) \bmod n$
3. Modular Addition: $s = (p + q) \bmod n$

4.1.1.1 Modular Exponentiation

Exponentiation involves solving the equation $x = a^e \bmod n$. Restrictions apply to each a, e, and n. The width of a and n, in bits, may be as large as 4096 bits, and the width of e must be less than or equal to 2048 bits. The value n must be odd.

The exponentiator core utilizes the Montgomery product algorithm, which contains a very important feature in that it involves a series of mathematical operations mod r, where r is a power of 2. The inclusion of the mod r function precludes the need for binary division, which can be costly in both gates and time. However, this technique requires the pre-calculation of constants derived from the original numbers a, e, and n. These constants are defined as:

$a'' = a * r \bmod n$

$x'' = r \bmod n$

$n' = (r * r^{-1} - 1)/n$

where: $r = 2^k$, $2^{k-1} <= n < 2^k$, and $r * r^{-1} \bmod n = 1$

Wherever possible, a”, x”, and n’ were precalculated as soon as a and n were available. The generation of these precalculated constants is computationally expensive to do in software, and decreases the overall performance of the system.

The AiMEC Montgomery Exponentiator Core is not an optimized mathematical core, thus all exponentiations with the same number of bits for the base and modulus will take the same amount of time. Each Montgomery exponentiation with a base and modulus of 512 bit, or less, and an exponent of 512 bits, or less, takes 2.1 mS.

4.1.1.2 Modular Multiplication and Addition

The multiplication operation as defined by $m = (p * q) \bmod n$, and addition operation as defined by $s = (p + q) \bmod n$, utilize an n whose bit width can be as large as 4096 bits. The length of p and q must be equal to or less than the bit width of n. It is not required that p and q be less than n in magnitude, nor is there a restriction that n be odd.

For each multiplication using a 384-bit modulus, the time to perform each multiplication is constant. According to [11], and collected timing, a modular multiplication with a modulus size of 512 bits, or less, takes 95 uS.

Another inefficiency of this embedment with the AiMEC Montgomery Exponentiator Core, is that a software modular reduction is necessary if the length of either operand (p or q), in bits, is greater than the length of the modulus (n), in bits. If p or q has a most significant bit of 1, and the most significant bit of n is 0, the results of the multiplication or addition are incorrect.

4.1.1.3 AiMEC Montgomery Exponentiator Core Summery

This AiMEC core is not an efficient mathematical core for performing fast elliptic curve operations. The necessity to precalculate values for an exponentiation or perform software modular reduction prior to an addition or multiplication is impractical. However, on this system, it is still faster than the software implementations for performing modular arithmetic.

Chapter 5 Finite Arithmetic Design and Test Setup

5.1. Finite Arithmetic written for the ARM946E

The AiMEC Montgomery Exponentiator Core is unable to perform non-modular multiplication, non-modular addition, and modular subtraction; C-code was used to perform these operations. While the C-code worked, it was not as efficient as necessary, and updates were made to allow for faster execution of the finite arithmetic.

For a modular multiplication, checks were added to determine if either of the input parameters is equal to one (since many of the z parameters could be equal to one). If so, a copy is performed in software rather than performing the modular multiplication using the AiMEC Montgomery Exponentiator. Checks were also added to determine if a modular reduction is necessary before attempting to perform the reduction in the modular subtraction routine.

The ARM946E contains the enhanced DSP capability of the ARM9E-S processor core which executes the extended ARM5vT instruction set, and includes the enhanced multiplier and saturating arithmetic functions. Writing assembly functions to perform the addition and subtraction of arrays, as well as 32-bit by 32-bit multiplies was better able to make use of the extended ARM5vT instruction set.

Included in the extended ARM5vT instruction set is the ability to perform a 32-bit by 32-bit multiply and obtain a 64 bit result with one instruction. By adding this instruction, plus the storage instructions to an assembly function, an entire C-routine was removed from the original finite arithmetic C-code. The assembly function, along with the description and inputs can be seen in Algorithm 5.

Algorithm 5: Assembly Routine `mult_32`

```
;******************************************************************************
;
; Description:  Performs the multiplication of two 32 bit unsigned integers
;               and stores the product.
;
;ARGUMENTS:       r0: Multiplier    ( 32 bits )
;                 r1: Multiplicand  ( 32 bits )
;                 r2: address of the Product  ( 64 bits )
;******************************************************************************
mult_32
    STMFD sp!, {r4-r11, lr}     ;save regs and return link

    UMULL   R3, R4, R0, R1

    STR     R3, [R2]           ; store the lower half
    ADD     R2, R2, #4         ; increment to next location
    STR     R4, [R2]           ; store the upper half

    LDMFD sp!, {r4-r11, pc}     ;restore regs and return
```

The other assembly functions written to handle the addition and subtraction of arrays can be found in Appendix A.1.

5.2. *Test Setup*

A diagram of the test setup is shown in Figure 8. The PC contained the ARM debugger and Multi-ICE software used to walk through and debug the code. The Multi-ICE interface unit attached the PC to the board. A logic analyzer was attached to a discrete hardware line that could be toggled during software execution.

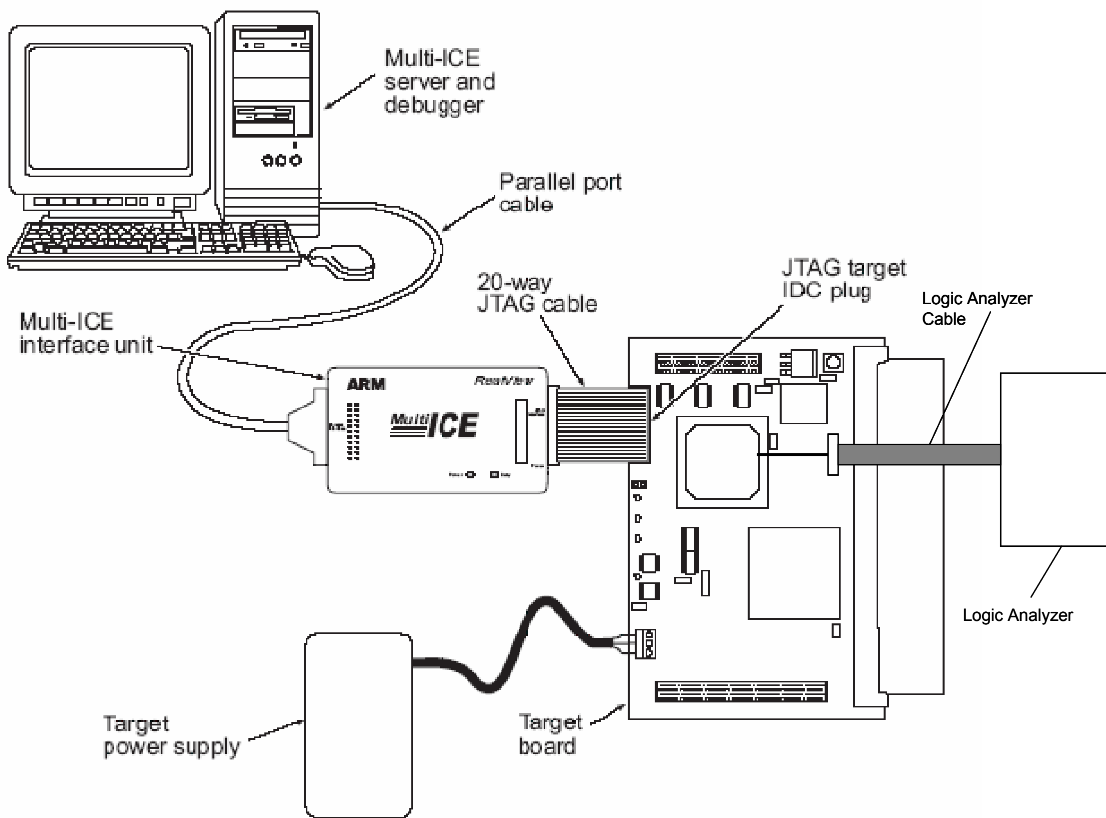

Figure 8: Test Setup Diagram
Image provided by ARM Limited

Chapter 6 Evaluation and Timing Descriptions

All of the timing measurements performed focused on a 384-bit elliptic curve multiplication when multiplying an elliptic curve point by a random scalar value. All measurements were taken using a toggled hardware line and measured with a logic analyzer as shown in Figure 8.

6.1. Baseline Timing Description

Based on the elliptic curve research, the elliptic curve addition function was written such that the inputs are one Jacobian and one Affine (refer back to section 3.2 for definition of Jacobian and Affine formats). The output is Jacobian as shown in Algorithm 1, i.e., P(x, y, z) + Q(x, y) = R(x, y, z). The elliptic curve double routine uses Jacobian coordinates for both the input and the output as shown in Algorithm 2, i.e., 2P(x, y, z) = R(x, y, z). The elliptic curve multiplication routine was written to utilize the Lem-Lee technique [8] with Shamir's technique [9] and three precalculated values as shown in Algorithm 4.

The first timing measurement of the 384-bit elliptic curve multiply routine was taken and on average took 600 mS to complete. This will be considered the baseline and can be described as follows:

- Algorithm 4 was used with a countermeasure to battle side channel attacks against timing.
- The existing finite arithmetic C-routines were used with no modifications.

- Instruction and data cache were both used.
- All of the code was placed in internal TCM.
- Most of the data was placed in internal TCM, with a limited amount in CRAM.
- The code was compiled using the Thumb instruction set.
- Compiler optimizations were set at the highest level for decreased code size.

6.2. Hardware Vs. Software Optimizations

With the baseline measurement established, the next step was a comparison to determine if non-modular multiplication and non-modular addition were faster to perform in software or utilizing the AiMEC Montgomery Exponentiator Core with a 384-bit modulus of all 1's. The software multiplication with the existing finite arithmetic C-routines was 10 times faster and software addition was 6 times faster than the exponentiator core with a modulus of all 1's. It was also determined that if more than 2 modular additions were needed, a single modular multiplication with the AiMEC Montgomery Exponentiator is faster, i.e. a modular multiply by 8 is faster than performing three modular additions. Updates were made to the elliptic curve code to use software whenever a modular reduction was not necessary, and to use modular multiplication using the AiMEC Montgomery Exponentiator whenever more than 2 modular additions were necessary. These updates decreased the elliptic curve multiply routine processing time by 51.5 mS and 8.6%, thus the 384-bit elliptic curve multiply took 548.5 mS on average. Figure 9, shows the original baseline elliptic curve multiply routine taking 600 mS as well as the elliptic curve multiply routine times following the

software updates to utilize the fastest approach. The time removed by the optimizations is included to keep the total execution time at 600 mS.

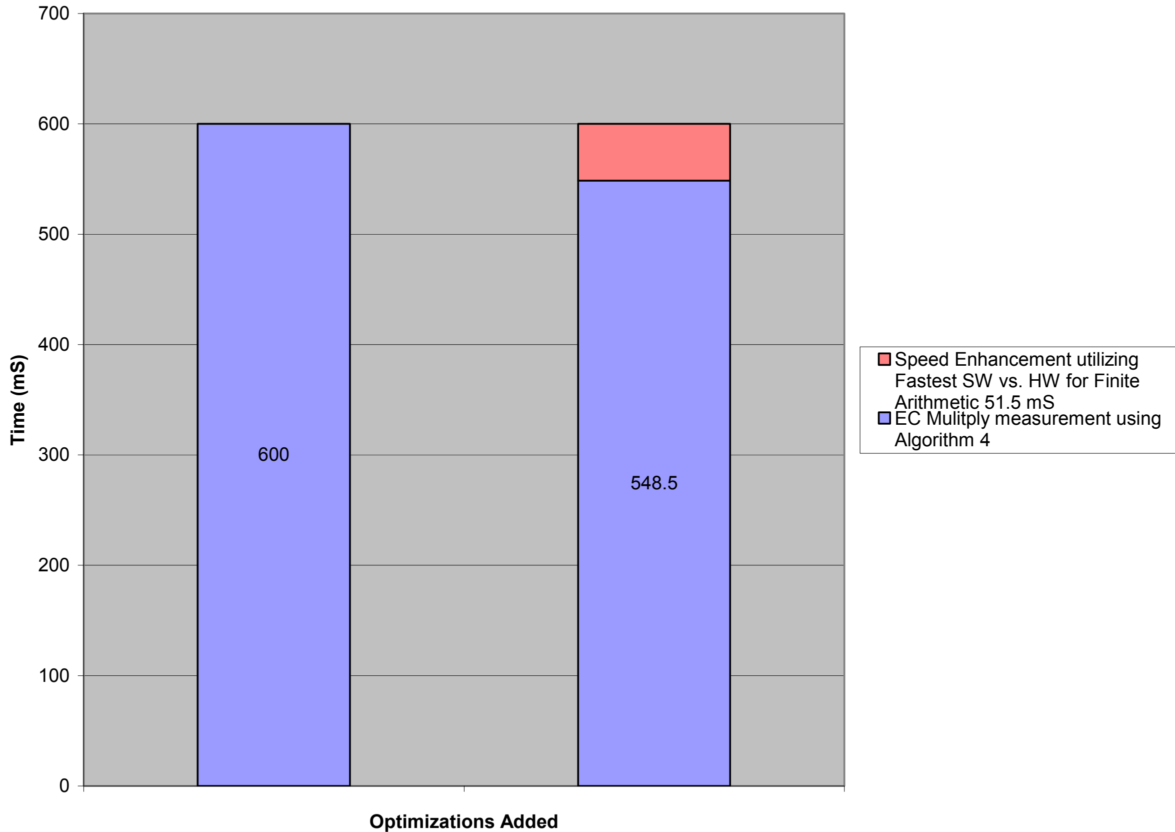

Figure 9: Elliptic Curve Multiply Baseline and HW/SW Optimization

6.3. Finite Arithmetic Software Optimizations

Multiple updates were next made to the finite arithmetic C-routines. First, assembly functions were written to perform the addition and subtraction of arrays as well as 32 x 32 bit multiplies. For a modular multiplication, checks were added to determine if either of the input parameters is equal to 1 (since many of the z parameters could be 1). If so, a copy is performed in software rather than performing the modular multiplication

using the AiMEC Montgomery Exponentiator. Checks were also included to determine if a modular reduction is necessary before attempting to perform the reduction in the modular subtraction routine. These updates removed an additional 80 mS and 14.59% from the 384-bit elliptic curve multiply execution time, bringing it to 468.5 mS on average. This is shown in Figure 9. This figure includes the previous updates to make use of the best software or hardware solution as well as the optimizations to the finite arithmetic. Each bar in the graph presents the total and component execution times for the baseline elliptic curve multiply routines prior to the optimizations.

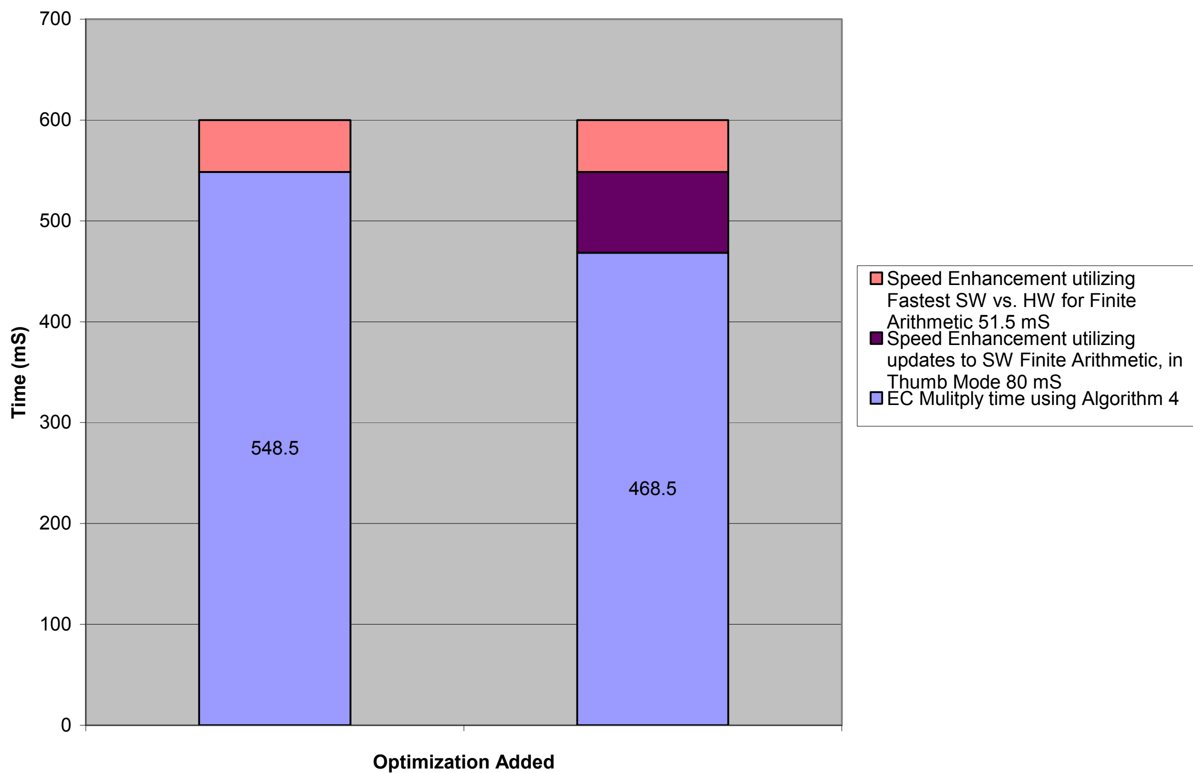

Figure 10: Elliptic Curve Multiply with Performance Optimizations 1 and 2

6.3.1 Finite Arithmetic Library Compiled in ARM Mode

The baseline timing measurement was performed with all the code compiled in the Thumb instruction set. Compiling the code with the ARM instruction set saved another 20 mS; however it increased the storage requirement for the code by more than 30% (about 4 Kbytes). The vast majority of the time spent performing an elliptic curve multiply is devoted to the finite arithmetic routines. Therefore, these routines were grouped into a separate "finite arithmetic" library and compiled using the ARM instruction set. The finite arithmetic library was then linked into the elliptic curve code and compiled using the Thumb instruction set. Given that the library was complied in ARM mode, and the elliptic curve was compiled in Thumb mode, the ARM/Thumb inter-working option was also included. The combination of the finite arithmetic library compiled in ARM, and the elliptic curve code compiled in Thumb, resulted in only an incremental increase to the code size and decreased the execution time by another 16.5 mS and 3.5%. The resulting execution time for the elliptic curve multiply routine was 452 mS on average. Figure 10, illustrates which enhancements thus far have had the greatest improvement to the timing.

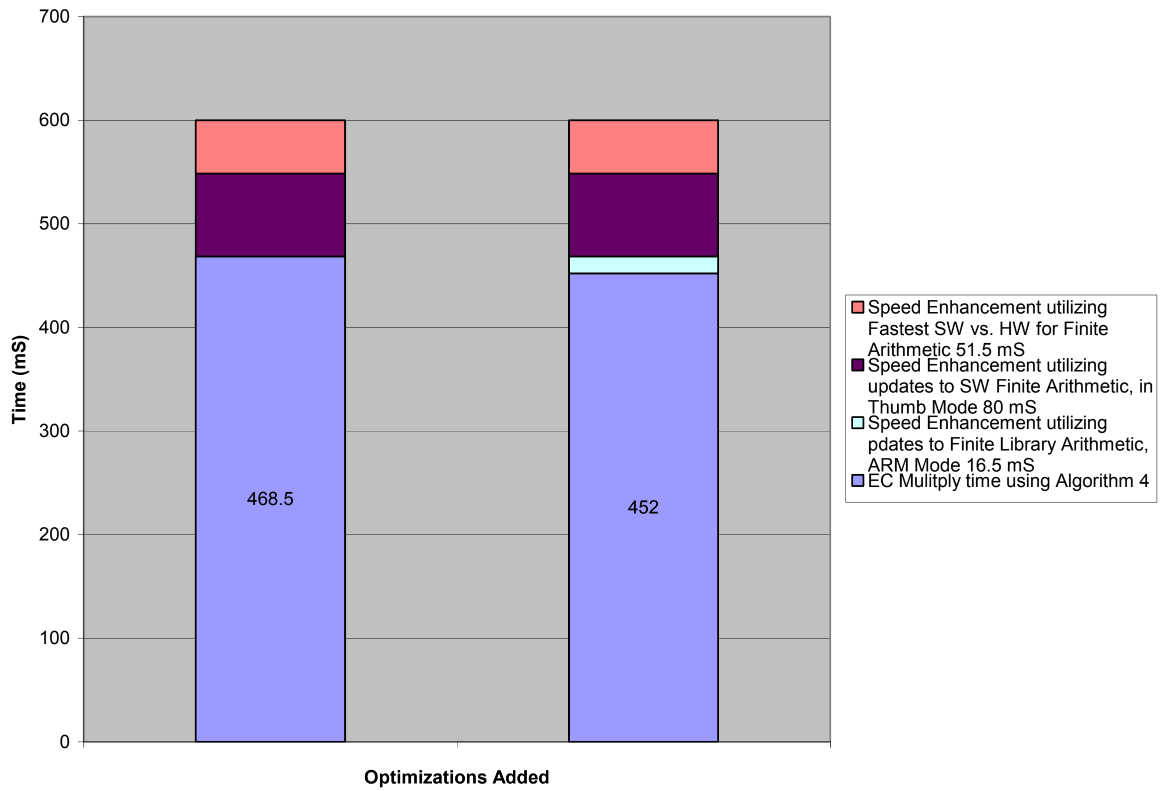

Figure 11: Elliptic Curve Multiply with Performance Optimizations 2 and 3

To verify that Algorithm 4 was a better alternative to Algorithm 3, the timing was investigated using the basic elliptic curve multiply routine (Algorithm 3). Here, no Lem-Lee technique [8], no Shamir's technique [9], and no pre-calculated values were used. The previous three updates described and shown in figures 8 – 10 are still included. The timing to perform the 384-bit elliptic curve multiply subsequently increased from 452 mS to 646 mS on average, a 30% increase in execution time.

6.4. Compiler Optimization Settings

To this point, all optimizations have been set to decrease the size of the code. Using Algorithm 4 with the countermeasure to battle side channel attacks, and the speed

enhancing updates shown in figures 8 -10; the optimizations were changed to increased execution speed, rather than decrease code size. Each of these enhancements, including the new optimization for speed, is depicted in Figure 11. The overall execution time to perform the 384-bit elliptic curve multiply was only decreased by another 6 mS and 1.3% from an average of 452 mS to an average of 446 mS.

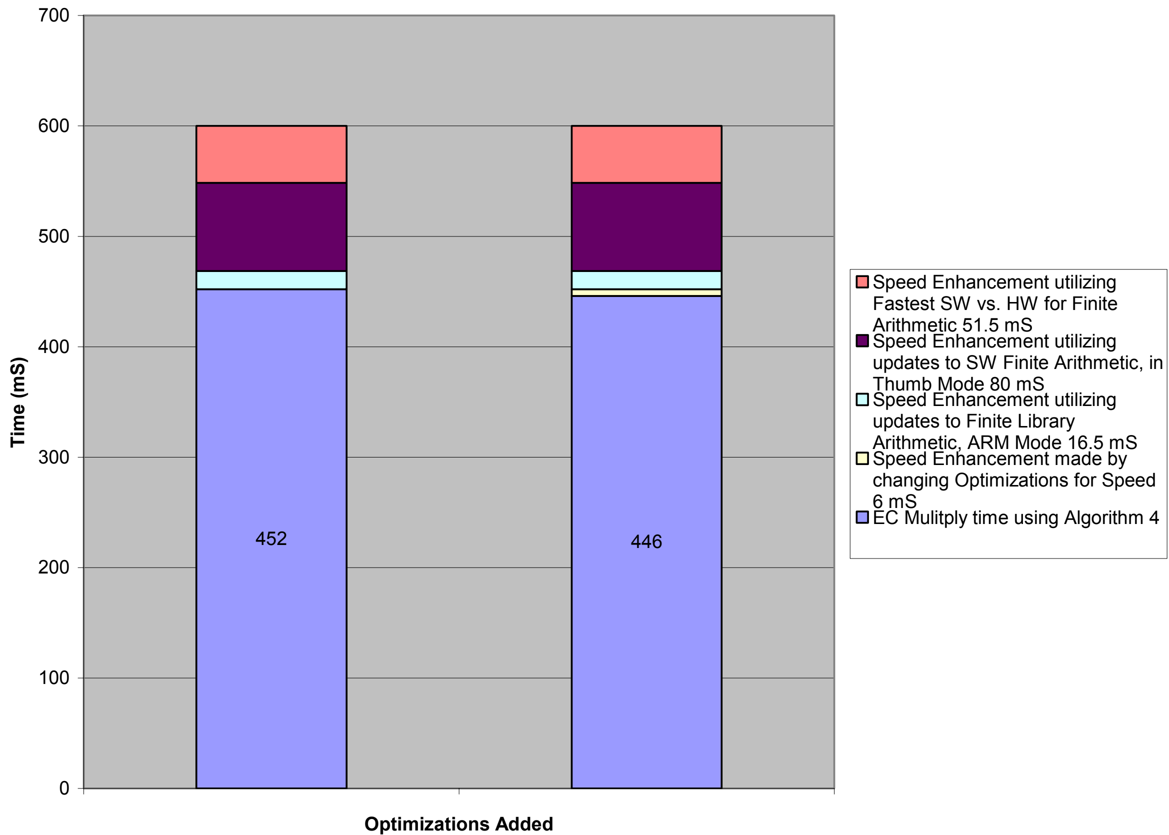

Figure 12: Elliptic Curve Multiply with Performance Optimizations 3 and 4

6.5. Side Channel Attacks Removed

Next, again using Algorithm 4, the countermeasure described earlier in this section for battling side channel attacks on timing, was removed. All other speed

enhancing updates described in the previous paragraphs of this section were included. The execution time to perform the 384-bit elliptic curve multiply was decreased by additional 70.5 mS on average to 375.5 mS, another 15.8% decrease.

6.6. Memory Options Investigated

To verify that TCM is faster than CRAM, all of the data and code were moved to CRAM rather than TCM. The time to perform the 384-bit elliptic curve multiply was increased by 5.6 mS from an average of 446 mS to 451.6 mS on average.

All timing measurements have been performed using the same cache setup. Instruction and data cache have been enabled for the CRAM. When the data cache was disabled, the time to complete the elliptic curve multiply routine increased to 472.4 mS on average. When both data and instruction cache were disabled, the average time measured was 473.3 mS. After the data cache was re-enabled and instruction cache was disabled, the time to complete the elliptic curve multiply routine took an average of 446.2 mS. These measurements show that the data cache had very little effect on system performance, and instruction cache had no impact on system performance. This can be attributed to the fact that TCM has no wait states and all of the code and nearly all of the data resided in TCM.

The first bar on the graph which is labeled Alg 4 Optimized, shown in Figure 12, illustrates the elliptic curve multiply algorithm performance after all optimizations were applied at an average of 446 mS. It is also shown in Figure 12 that the time to execute the algorithm was increased when the caching options were changed, or the code is moved from TCM to CRAM, which is not resident on the ARM processor. The last item

shown in Figure 12 is the amount of time increase to the optimized algorithm performance when the counter measures are included.

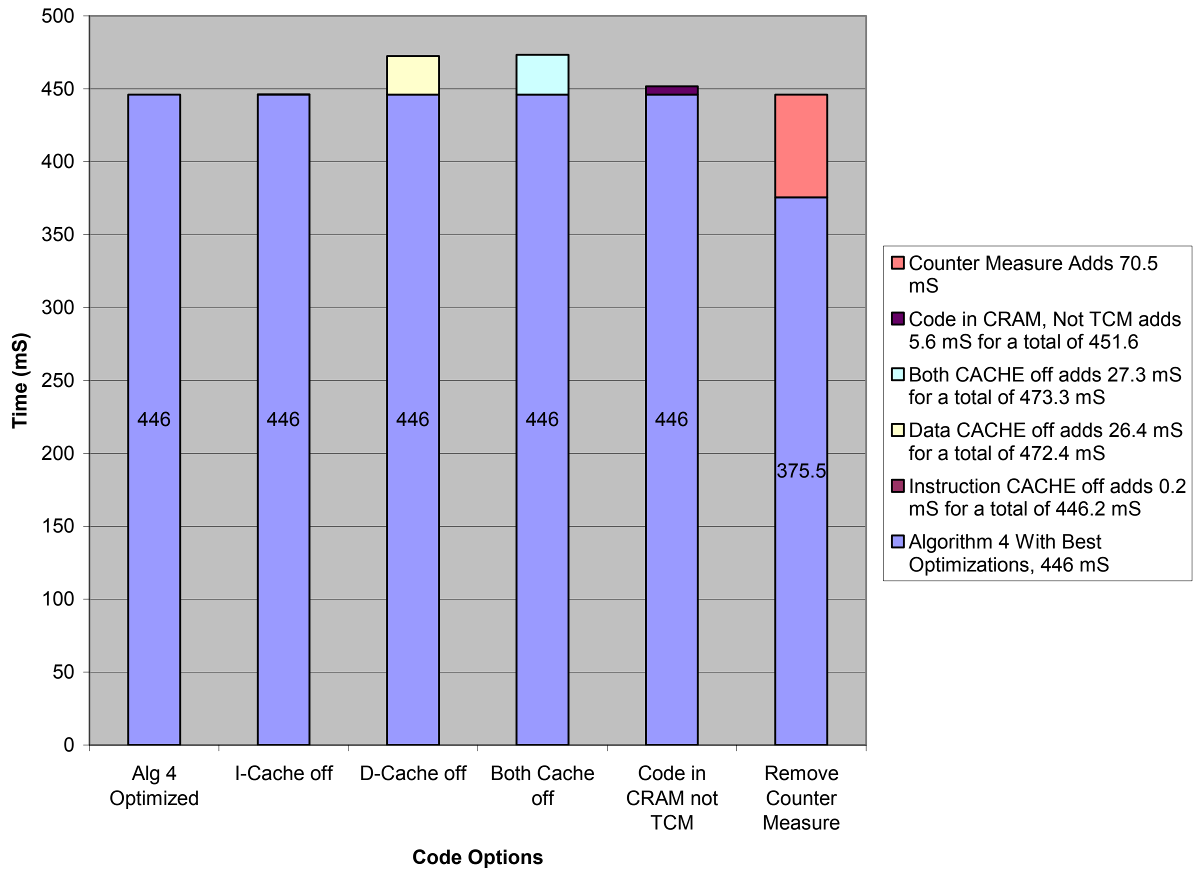

Figure 13: Other Code Options

Chapter 7 Summary and Future Work

Through the empirical timing measurements, it has been shown that the largest decrease in execution time was due to enhancements made in the finite arithmetic functions. These enhancements provided a 22% time reduction in a 384-bit elliptic curve multiplication. These enhancements included:

- Rewriting a portion of the finite arithmetic to use assembly functions that perform the addition and subtraction of arrays and 32 x 32 bit multiplies.
- Adding checks prior to performing a modular reduction.
- Using the Exponentiator Core only when modulus reduction was necessary.
- Using multiplication if more than two additions would be required.
- Placing the finite arithmetic into its own library and using ARM mode.

Based on the requirements of the system, the side channel attacks could also be removed. In addition to the 22% decrease in execution time from the previous enhancements, another 15% reduction in execution time was attained when the countermeasure for the side channel attack on timing was not implemented.

The other optimizations investigated including: cache usage, compiler options (speed vs. size), and Thumb instruction set vs. ARM instruction set provided minimal decrease in the execution time of 3.6%.

All of the enhancements implemented can be seen collectively in Figure 13. The baseline elliptic curve multiply routine is shown using algorithm 4 followed by each enhancement made. The decrease in time from the baseline measurement for each optimization is depicted using a different color. Once more, it can be seen by the graph in Figure 13 that the updates made in the finite arithmetic functions provided the most reduction in execution time.

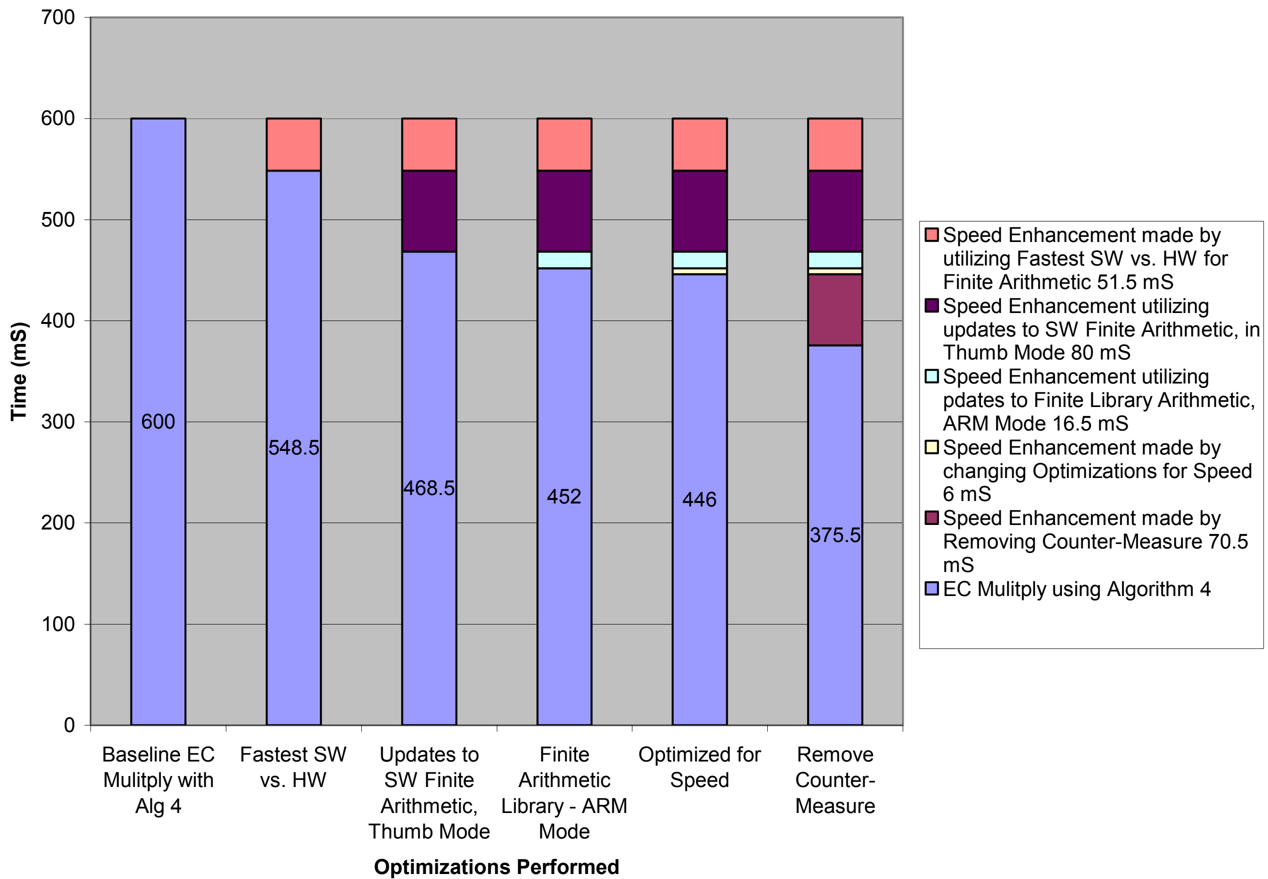

Figure 14: Finite Arithmetic and ARM Optimization Used

Table 1 provides a different format to show an overview of all options tested and the timing achieved. The light blue line indicates the baseline system described in the third paragraph of the evaluation and timing descriptions. The green lines indicate the most efficient option found on this system. The difference between the two options is the use of the countermeasure for a time side channel attack. The percentages indicate the difference from the base system.

384-bit EC Multiply Time average (mS)	Algorithm 4	Algorithm 3	Fastest SW vs. HW	Updates to Finite Arithmetic Thumb Mode	Finite Arithmetic library ARM	Compiler options set for Time	Compiler options set for code size	Code and data in TCM	Code and data in CRAM	ICache used	DCache Used	Timing Counter Measure	% speed improvement over baseline
646		**X**	**X**	**X**	**X**		**X**	**X**		**X**	**X**	**X**	**-7.6**
600	**X**						**X**	**X**		**X**	**X**	**X**	**--**
548.5	**X**		**X**				**X**	**X**		**X**	**X**	**X**	**8.6**
468	**X**		**X**	**X**			**X**	**X**		**X**	**X**	**X**	**22**
452	**X**		**X**	**X**	**X**		**X**	**X**		**X**	**X**	**X**	**24.67**
446	**X**		**X**	**X**	**X**	**X**		**X**		**X**	**X**	**X**	**25.66**
375.5	**X**		**X**	**X**	**X**	**X**		**X**		**X**	**X**		**37.42**
451.6	**X**		**X**	**X**	**X**	**X**			**X**	**X**	**X**	**X**	**24.29**
472.4	**X**		**X**	**X**	**X**	**X**		**X**		**X**		**X**	**21.27**
473.3	**X**		**X**	**X**	**X**	**X**		**X**				**X**	**21.12**
446.2	**X**		**X**	**X**	**X**	**X**		**X**			**X**	**X**	**25.66**

Table 1: Overview of all options tested

Future work that could be explored includes:

- A hardware, FPGA or ASIC, version of the elliptic curve and finite arithmetic routines.

- Usage of an optimized mathematical core that does not return a constant time for mathematical operations.
- Using more precalculated values.

Bibliography

[1] Certicom Corp. *"Elliptic* Curve *Cryptography Tutorial*", 4 May. 2007. Reference .com http://www.certicom.com/index.php?action=ecc_tutorial,home

[2] M. Brown, D. Hankerson, J.Lopez, and A. Menezes. "*Software Implementation of the NIST Elliptic Curves Over Prime Fields*"

[3] M. Hopper. "Mathematical *Routines for Elliptic Curves*", 21 February. 2001

[4] Y. Hitchcodk, E. Dawson, A. Clark, P. Montague. "*Implementing an efficient elliptic curve crypto system over GF(p) on a smart card*", 24 October. 2002

[5] H. Cohen, A. Miyaji, T. Ono. *"Efficient elliptic curve exponentiation using mixed coordinates"* (http://citeseer.ist.psu.edu/277895.html), ASIACRYPT 1998

[6] B. Henhapl. *"*Platform *Independent Elliptic Curve Cryptography over Fp"*

[7] NIST, *"*Recommendation *for Key Management —Part 1:general"* (http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf), Special Publication 800-57, August 2005

[8] C. Lim, P. Lee. "*More Flexible Exponentiation with precomputation*", Advances in Cryptology, Proceedings of Crypto '94, Volume 839, pages 95-107, 1994

[9] J.Solinas, *"Shamir's Trick for Elliptic Curves"*, Version 328, 28 March. 2000

[10] ARM Limited, *"ARM Online Books"*, ARM Developers Suite V1.2

[11] Flextronics Semiconductor, "*AiMEC Montgomery Exponentiator Core"*, Specification number 12016-0315

[12] ARM Limited, *"Architecture Reference Manual"*, Second Edition 2000

[13] A. Sloss, D. Symes, C. Wright, *"ARM System Developer's Guide Designing and Optimizing System Software"* 2004

[i] RSA is an algorithm for public-key cryptography. It was the first algorithm known to be suitable for signing as well as encryption. The algorithm was publicly described in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman at MIT; the letters RSA are the initials of their surnames.

Appendix A

A.1 Finite Arithmetic Assembly Routines

```
;***************************************************************************
;
;FUNCTION:       mult_32
;
;LANGUAGE:      ARM946E-S Assembly
;
; Description:  Performs the multiplication of two 32 bit unsigned integers
;               and stores the product.
;
;ARGUMENTS:       r0: Multiplier    ( 32 bits )
;                 r1: Multiplicand  ( 32 bits )
;                 r2: address of the Product  ( 64 bits )
;*****************************************************************************
    AREA Utility, CODE, READONLY
    CODE32


    EXPORT mult_32

mult_32
    STMFD sp!, {r4-r11, lr}     ;save regs and return link

    UMULL   R3, R4, R0, R1

    STR     R3, [R2]           ; store the lower half
    ADD     R2, R2, #4         ; increment to next location
    STR     R4, [R2]           ; store the upper half

    LDMFD sp!, {r4-r11, pc}     ;restore regs and return
```

```
;*************************************************************************
;
;FUNCTION:       add_arrays
;
;LANGUAGE:       ARM946E-S Assembly
;
; Description:  Performs the addition of two arrays and stores the sum.  Also
;               increments the length if a carry occurs.
;
;ARGUMENTS:      r0: Address of the addend
;                r1: Address of the augend
;                r2: Address of the sum
;                r3: Address of the number of words to add
;                      also number of words of the result
;***************************************************************************
    EXPORT add_arrays


add_arrays
    STMFD sp!, {r4-r11, lr}     ;save regs and return link
    LDR R8, [R3]            ; place the number of words to add into R8
    MOV R7, #0              ; Zeroize R7 to use to clear and restore the CPSR

add_loop
    MSR CPSR_f, R7       ; clear all the flags( 1st time ) OR restore from ADD

    LDR R4, [R0], #4     ; place a word from the addend into R4, increment R0
(address)
    LDR R5, [R1], #4     ; place a word from the augend into R5, increment R1

    ADCS  R6, R4, R5     ; add augend, addend and Carry, place in R6, set flags
    STR   R6, [R2], #4   ; store the result into the sum, increment the sum ptr

    MRS R7, CPSR         ;save the CPSR if to resore before add

    SUBS R8, R8, #1      ; Decrement the counter
    BNE add_loop         ; loop until complete (z flag not set)

    MSR CPSR_f, R7       ; restore flags from ADD
    BCC add_done         ; Branch on no carry (carry clear)

                         ; other wise handle the carry
    LDR R8, [R3]         ; reload the number of words to add into R8
    ADD R8, R8, #1       ; update the number of words of the result
    STR R8, [R3]         ; store the number of words of the result

add_done
    LDMFD sp!, {r4-r11, pc}     ;restore regs and return
```

```
;***************************************************************************
;
;FUNCTION:       add_1_to_array
;
;LANGUAGE:       ARM946E-S Assembly
;
; Description:  Adds 1 to an array and stores the sum.  Also
;               increments the length if a carry occurs.
;
;ARGUMENTS:      r0: Address of the array
;                r1: Address of the sum
;                r2: Address of the number of words to add
;                      also number of words of the result
;****************************************************************************
    EXPORT add_1_to_array


add_1_to_array
    STMFD sp!, {r4-r11, lr}     ;save regs and return link
    LDR R8, [R2]            ; place the number of words to add into R8
    MOV R7, #0              ; Zeroize R7 to use to clear and restore the CPSR
    MOV R5, #1              ; place a 1 into R5 for the 1st add

loop
    MSR CPSR_f, R7       ; clear all the flags( 1st time ) OR restore from ADD

    LDR R4, [R0], #4     ; place a word from the array into R4, increment R0
(address)

    ADCS  R6, R4, R5     ; add word from array + 0 or 1 + Carry, place in R6, set
flags
    STR   R6, [R1], #4   ; store the result into the sum, increment the sum ptr

    MRS R7, CPSR         ;save the CPSR if to resore before add
    MOV R5, #0           ; place a 0 into R5 for the remaining adds

    SUBS R8, R8, #1      ; Decrement the counter
    BNE loop             ; loop until complete (z flag not set)

    MSR CPSR_f, R7       ; restore flags from ADD
    BCC done             ; Branch on no carry (carry clear)

                         ; other wise handle the carry
    STR R5, [R1]         ; store 1 into the sum
    LDR R8, [R2]         ; reload the number of words to add into R8
    ADD R8, R8, #1       ; update the number of words of the result
    STR R8, [R2]         ; store the number of words of the result

done
    LDMFD sp!, {r4-r11, pc}     ;restore regs and return
```

```
;**************************************************************************
;
;FUNCTION:       subtract_arrays
;
;LANGUAGE:       ARM946E-S Assembly
;
; Description:  Performs the subtraction of two arrays and stores the difference,
;               also increments the length if a carry occurs.
;
;ARGUMENTS:      r0: Address of the minuend
;                r1: Address of the subtractend
;                r2: Address of the difference
;                r3: Address of the number of words to subtract
;
;****************************************************************************
    EXPORT subtract_arrays


subtract_arrays
    STMFD sp!, {r4-r11, lr}     ;save regs and return link
    LDR R8, [R3]            ; place the number of words to add into R8
    MOV R7, #0x20000000     ; bit 29 of the CPSR is the Carry bit, needs
                            ; to be set to 1 to indicate no barrow

sub_loop
    MSR CPSR_f, R7       ; update the flags( 1st time ) OR restore from Sub

    LDR R4, [R0], #4     ; place a word from the minuend into R4, increment R0
(address)
    LDR R5, [R1], #4     ; place a word from the subtractend into R5, increment
R1

    SBCS  R6, R4, R5     ; add sub with carry, place in R6, set flags
    STR   R6, [R2], #4   ; store the result into the difference, increment the
diff ptr

    MRS R7, CPSR         ;save the CPSR if to resore before subtract

    SUBS R8, R8, #1      ; Decrement the counter
    BNE sub_loop         ; loop until complete (z flag not set)

    MSR CPSR_f, R7       ; restore flags from SUB
    BCS sub_done         ; Branch on no Barrow (carry set)

                         ; other wise handle the carry
    LDR R8, [R3]         ; reload the number of words to subtract into R8
    SUB R8, R8, #1       ; decrement to indicate cary,
    STR R8, [R3]         ; store the number of words of the result

sub_done
    LDMFD sp!, {r4-r11, pc}     ;restore regs and return
```

```
;************************************************************************
;
;FUNCTION:       sub_1_from_array
;
;LANGUAGE:       ARM946E-S Assembly
;
; Description:  Adds 1 to an array and stores the sum.  Also
;               increments the length if a carry occurs.
;
;ARGUMENTS:      r0: Address of the array
;                r1: Address of the diff
;                r2: Address of the number of words to sub
;                      also number of words of the result
;**************************************************************************
    EXPORT sub_1_from_array


sub_1_from_array
    STMFD sp!, {r4-r11, lr}     ;save regs and return link
    LDR R8, [R2]            ; place the number of words to sub into R8
    MOV R7, #0x20000000     ; bit 29 of the CPSR is the Carry bit, needs
                            ; to be set to 1 to indicate no barrow
    MOV R5, #1              ; place a 1 into R5 for the 1st subtract

sub1_loop
    MSR CPSR_f, R7       ; clear all the flags( 1st time ) OR restore from ADD

    LDR R4, [R0], #4     ; place a word from the array into R4, increment R0
(address)

    SBCS  R6, R4, R5     ; sub word from array - 0 or 1 - Carry, place in R6, set
flags
    STR   R6, [R1], #4   ; store the result into the diff, increment the diff ptr

    MRS R7, CPSR         ; save the CPSR if to resore before sub
    MOV R5, #0           ; place a 0 into R5 for the remaining sub

    SUBS R8, R8, #1      ; Decrement the counter
    BNE sub1_loop        ; loop until complete (z flag not set)

                         ; there should never be a carry on the final subtract
                         ; since we already made sure the orig number was
                         ; smaller.


;sub1_done
    LDMFD sp!, {r4-r11, pc}     ;restore regs and return



 END
```

A.2 Elliptic Curve C Routines

```
/*****************************************************************************

FILE:          Elliptic_Curve_Module.c

PACKAGE:       Elliptic_Curve_Module

LANGUAGE:      ANSI C

UNIT TEST:     Elliptic_Curve_Module_Unit_Test

REV. HISTORY:

*****************************************************************************/

/***************************************/
/* INCLUDE FILES                       */
/***************************************/
#include "common.h"

#include <string.h>

#include "burst_utility.h"
#include "Elliptic_Curve_Module.h"
#include "Elliptic_Curve_Types.h"
#include "Firefly_External_Types.h"
#include "Firefly_Utilities.h"
#include "MPA.h"
#include "MPA_Multiplier_Chip.h"
#include "MPA_Types.h"

/***************************************/
/* INTERNAL MACRO DEFINITIONS          */
/***************************************/
// None

/***************************************/
/* INTERNAL TYPE DEFINITIONS           */
/***************************************/
// None

/***************************************/
/* PRIVATE FUNCTION DECLARATIONS       */
/***************************************/

static void _EC_Add(
    t_PROJECTIVE_POINT*     Point_S_Ptr,
    t_PROJECTIVE_POINT*     Point_T_Ptr,
    t_DATA_AND_SIZE_STRUCT* Modulus_Ptr,
    t_PROJECTIVE_POINT*     Point_R_Ptr);

/***************************************/
/* PUBLIC MEMBER VARIABLES             */
/***************************************/
//None

/***************************************/
/* PRIVATE MEMBER VARIABLES            */
/***************************************/
```

```
static const t_UINT32 ADD_PAD_SIZE_WORDS         =  1;

// these need to be static so that they are not placed on the stack
static t_UINT32 fg_tempBuffer1[ MAX_COORDINATE_SIZE_WORDS ];
static t_UINT32 fg_tempBuffer2[ MAX_COORDINATE_SIZE_WORDS ];
static t_UINT32 fg_tempBuffer3[ MAX_COORDINATE_SIZE_WORDS ];
static t_UINT32 fg_tempBuffer4[ MAX_COORDINATE_SIZE_WORDS ];
static t_UINT32 fg_tempBuffer5[ MAX_COORDINATE_SIZE_WORDS ];
static t_UINT32 fg_tempBuffer6[ MAX_COORDINATE_SIZE_WORDS + ADD_PAD_SIZE_WORDS ];
static t_UINT32 fg_tempBuffer7[ MAX_COORDINATE_SIZE_WORDS ];
static t_UINT32 fg_tempBuffer8[ MAX_COORDINATE_SIZE_WORDS + ADD_PAD_SIZE_WORDS ];
static t_UINT32 fg_tempBuffer9[ MAX_COORDINATE_SIZE_WORDS + ADD_PAD_SIZE_WORDS ];

/**************************************/
/* PUBLIC FUNCTION DEFINITIONS        */
/**************************************/


/****************************************************************************

FUNCTION:        EC_Convert_Proj_To_Affine

DESCRIPTION:
         EC_Convert_Proj_To_Affine accepts a projective point S, and
         returns an affine point R.  The Rx portion becomes x/(z^2), the
         Ry portion becomes y/(z^3).  Z^(-1) can be calculated as Z^(p-2) Mod
         p.  This equation is valid when the modulus (p) is prime, which is
         true for Elliptic Curves.

ARGUMENTS:
    Point_S_Ptr -
             Points to the projective point to convert.

    Modulus_Ptr -
            Points to the modulus that is used during the conversion of a
            projective point to an affine point.

    Modulus_Size_Bytes -
            The length of the modulus in bytes.

    n_Prime_For_Modulus_Ptr -
            The pre-calculated n' value for the passed in modulus.  Used during
            exponentiate math operation.

    n_Prime_For_Modulus_Length_Bytes -
            The length of the pre-calcualted n' value for the passed in
            modulus.  Used during the exponentiate math operation.

    Point_R_Ptr -
            The affine point resulting from the conversion of the passed
            in projective point is returned in this parameter.

RETURN VALUES:   None

LIMITATIONS:     None

NOTES:           None

****************************************************************************/
void EC_Convert_Proj_To_Affine( t_PROJECTIVE_POINT*     Point_S_Ptr,
                                t_DATA_AND_SIZE_STRUCT* Modulus_Ptr,
                                t_DATA_AND_SIZE_STRUCT* n_Prime_For_Modulus_Ptr,
                                t_AFFINE_POINT*         Point_R_Ptr )
```

```
{
    t_MPA_BIG_INT_STRUCT MPA_Sub_Minuend;
    t_MPA_BIG_INT_STRUCT MPA_Sub_Subtrahend;
    t_MPA_BIG_INT_STRUCT MPA_Sub_Difference;

    t_MPA_BIG_INT_STRUCT MPA_Mod_Mult_Multiplier;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Mult_Multiplicand;
    t_MPA_BIG_INT_STRUCT MPA_Modulus;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Mult_Product;

    t_MPA_BIG_INT_STRUCT MPA_Exp_Base;
    t_MPA_BIG_INT_STRUCT MPA_Exp_Exponent;
    t_MPA_BIG_INT_STRUCT MPA_Exp_N_Prime;
    t_MPA_BIG_INT_STRUCT MPA_Exp_Result;

    t_MPA_EXPONENTIATE_STRUCT MPA_Exp_Struct;

    t_UINT32 temp_variable = 2; // needed for subtracting and multiplying


    // set up the modulus this will never change.
    MPA_Modulus.Data_Length = Modulus_Ptr->data_length_words;
    MPA_Modulus.Data_Ptr = Modulus_Ptr->data_ptr;

    // the length of the mulitplicand and multiplier are always the same
    MPA_Mod_Mult_Multiplier.Data_Length = MAX_COORDINATE_SIZE_WORDS;
    MPA_Mod_Mult_Multiplicand.Data_Length = MAX_COORDINATE_SIZE_WORDS;


    // Step 1
    // Calculate 1/z = z^(p - 2) Mod p and save as inverse Z Value by performing
    //  the following steps:
     MPA_Sub_Minuend.Data_Length = Modulus_Ptr->data_length_words;
     MPA_Sub_Minuend.Data_Ptr    = Modulus_Ptr->data_ptr;

     MPA_Sub_Subtrahend.Data_Length = SIZE_WORDS( t_UINT32 );
     MPA_Sub_Subtrahend.Data_Ptr = &temp_variable;  // already set to 2

     MPA_Sub_Difference.Data_Ptr = &fg_tempBuffer3[0];

     (void) MPA_Subtract ( &MPA_Sub_Minuend,       // p
                           &MPA_Sub_Subtrahend,    // 2
                           &MPA_Sub_Difference );  // p - 2

     MPA_Exp_Base.Data_Length = MAX_COORDINATE_SIZE_WORDS;
     MPA_Exp_Base.Data_Ptr = Point_S_Ptr->Z_Ptr;

     MPA_Exp_Exponent.Data_Length = SIZE_WORDS( fg_tempBuffer3 );
     MPA_Exp_Exponent.Data_Ptr = &fg_tempBuffer3[ 0 ];

     MPA_Exp_N_Prime.Data_Length = n_Prime_For_Modulus_Ptr->data_length_words;
     MPA_Exp_N_Prime.Data_Ptr = n_Prime_For_Modulus_Ptr->data_ptr;

     MPA_Exp_Result.Data_Ptr = &fg_tempBuffer1[ 0 ];

     MPA_Exp_Struct.Base_Ptr = &MPA_Exp_Base;           // Z
     MPA_Exp_Struct.Exponent_Ptr = &MPA_Exp_Exponent;   // p - 2
     MPA_Exp_Struct.Modulus_Ptr = &MPA_Modulus;     // p
     MPA_Exp_Struct.N_Prime_Ptr = &MPA_Exp_N_Prime;
     MPA_Exp_Struct.Result_Ptr = &MPA_Exp_Result;       // result = (1/z)

     (void) MPA_Mod_Exponentiate ( &MPA_Exp_Struct );
```

```
    // Step 2
    // Calculate (1/z)^2 and save as temp2 by performing the following steps:
    MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer1[ 0 ];

    MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer1[ 0 ];

    MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer2[0];

    (void) MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier,
                              &MPA_Mod_Mult_Multiplicand,
                              &MPA_Modulus,
                              &MPA_Mod_Mult_Product );  //  (1/z) ^ 2

    // Step 3
    // Calculate Sx * ((1/z) ^ 2) and save as Rx by performing the following
    //  steps:
    MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer2[ 0 ];

    MPA_Mod_Mult_Multiplicand.Data_Ptr = Point_S_Ptr->XY_Ptr->X_Ptr;

    MPA_Mod_Mult_Product.Data_Ptr = Point_R_Ptr->X_Ptr;


    (void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // (1/z)^
                             &MPA_Mod_Mult_Multiplicand, // Sx
                             &MPA_Modulus,
                             &MPA_Mod_Mult_Product ); // Sx * ((1/z) ^ 2)

    // Step 4
    // Calculate ((1/z) ^ 2) * 1/z and save in a result buffer.

    // multipler is still temp 2
    // length of the multiplicand has not changed.
    MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer1[0];

    MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer3[0];

    (void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier,
                             &MPA_Mod_Mult_Multiplicand,
                             &MPA_Modulus,
                             &MPA_Mod_Mult_Product ); //result = (1/z) ^3

    // Step 5
    // Calculate ((1/z) ^3) * Sy and save as Ry.
    MPA_Mod_Mult_Multiplier.Data_Length = SIZE_WORDS( fg_tempBuffer3 );
    MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer3[ 0 ];

    // length of the multiplicand has not changed.
    MPA_Mod_Mult_Multiplicand.Data_Ptr = Point_S_Ptr->XY_Ptr->Y_Ptr;

    MPA_Mod_Mult_Product.Data_Ptr = Point_R_Ptr->Y_Ptr;

    (void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // (1/z)^3
                             &MPA_Mod_Mult_Multiplicand, // Sy
                             &MPA_Modulus,
                             &MPA_Mod_Mult_Product );// (1/z)^3 ) * Sy
    return;
}
```

```
/*****************************************************************************

FUNCTION:        EC_Double

DESCRIPTION:
         EC_Double accepts 1 projective point S, and returns a projective
         point R such that R = 2S.

ARGUMENTS:
    Point_S_Ptr -
            Points to the projective point to add to itself (is equivalent
            to multiplying the projective point by 2).

    Modulus_Ptr -
            Points to the modulus, which is used during the Elliptic Curve
            double operation.

    Modulus_Size_Bytes -
            The length of the modulus in bytes.

    Point_R_Ptr -
            The projective point resulting, from the Elliptic Curve doubling
            of the passed in projective point, is returned in this parameter.

RETURN VALUES:   None

LIMITATIONS:     None

NOTES:           None

*****************************************************************************/
void EC_Double( t_PROJECTIVE_POINT*     Point_S_Ptr,
                t_DATA_AND_SIZE_STRUCT* Modulus_Ptr,
                t_PROJECTIVE_POINT*     Point_R_Ptr )
{
    t_MPA_BIG_INT_STRUCT MPA_Mod_Sub_Minuend;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Sub_Subtrahend;
    t_MPA_BIG_INT_STRUCT MPA_Modulus;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Sub_Difference;

    t_MPA_BIG_INT_STRUCT MPA_Mod_Add_Augend;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Add_Addend;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Add_Sum;

    t_MPA_BIG_INT_STRUCT MPA_Mod_Mult_Multiplier;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Mult_Multiplicand;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Mult_Product;


    // Step 1
    if ( ( memcmp ( Point_S_Ptr->XY_Ptr->Y_Ptr,
                    &g_ZEROES[0],                // Initialized to 0
                    MAX_COORDINATE_SIZE_BYTES ) == 0 ) ||

         ( memcmp ( Point_S_Ptr->Z_Ptr,
                    &g_ZEROES[0],                // Initialized to 0
                    MAX_COORDINATE_SIZE_BYTES ) == 0 ) )
    {
        // Set Point_R_Ptr to
        //    Rx = 1, Ry = 1, & Rz = 0.

        // Set Rx to all zeroes.
        Burst_Fill ( Point_R_Ptr->XY_Ptr->X_Ptr,
```

```
                &g_ZEROES[ 0 ],
                MAX_COORDINATE_SIZE_WORDS );

    // Now set Rx to 1.
    *(Point_R_Ptr->XY_Ptr->X_Ptr) = 1;

    // Set Ry to all zeroes.
    Burst_Fill ( Point_R_Ptr->XY_Ptr->Y_Ptr,
                &g_ZEROES[ 0 ],
                MAX_COORDINATE_SIZE_WORDS );

    // Now set Ry to 1.
    *(Point_R_Ptr->XY_Ptr->Y_Ptr) = 1;

    // Set Rz to all zeroes.
    Burst_Fill ( Point_R_Ptr->Z_Ptr,
                &g_ZEROES[ 0 ],
                MAX_COORDINATE_SIZE_WORDS );
}

else
{
    // set up the modulus and data lengths and that will not change
    MPA_Modulus.Data_Length = Modulus_Ptr->data_length_words;
    MPA_Modulus.Data_Ptr = Modulus_Ptr->data_ptr;

    MPA_Mod_Mult_Multiplier.Data_Length = MAX_COORDINATE_SIZE_WORDS;
    MPA_Mod_Mult_Multiplicand.Data_Length = MAX_COORDINATE_SIZE_WORDS;

    MPA_Mod_Sub_Minuend.Data_Length = MAX_COORDINATE_SIZE_WORDS;
    MPA_Mod_Sub_Subtrahend.Data_Length = MAX_COORDINATE_SIZE_WORDS;

    MPA_Mod_Add_Augend.Data_Length = MAX_COORDINATE_SIZE_WORDS;
    MPA_Mod_Add_Addend.Data_Length = MAX_COORDINATE_SIZE_WORDS;

    // Step 2
    // Set tempBuffer1 equal to Sx (x portion of Point_S_Ptr).
    Burst_Copy ( &fg_tempBuffer1[ 0 ],          // copy to
                Point_S_Ptr->XY_Ptr->X_Ptr,     // copy from
                MAX_COORDINATE_SIZE_WORDS );

    // Set tempBuffer2 equal to Sy (y portion of Point_S_Ptr).
    Burst_Copy ( &fg_tempBuffer2[ 0 ],          // copy to
                Point_S_Ptr->XY_Ptr->Y_Ptr,     // copy from
                MAX_COORDINATE_SIZE_WORDS );

    // Set tempBuffer3 equal to Sz (z portion of Point_S_Ptr).
    Burst_Copy ( &fg_tempBuffer3[ 0 ],          // copy to
                Point_S_Ptr->Z_Ptr,             // copy from
                MAX_COORDINATE_SIZE_WORDS );


    // Step 3
    // Calculate tempBuffer3 (Sz) ^ 2 and save to tempBuffer4.
    MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer3[ 0 ];

    MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer3[ 0 ];

    MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer4[ 0 ];

    (void) MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier,   // Sz
                              &MPA_Mod_Mult_Multiplicand, // Sz
                              &MPA_Modulus,
```

```
                          &MPA_Mod_Mult_Product );     // result =
                                                       //      Sz * Sz

// Step 4
// Subtract tempBuffer4 (Sz ^ 2) from tempBuffer1 (Sx)
//  and store in tempBuffer5.
MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer1[ 0 ];

MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer4[ 0 ];

MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer5[ 0 ];

(void) MPA_Mod_Subtract( &MPA_Mod_Sub_Minuend,      // Sx
                         &MPA_Mod_Sub_Subtrahend,   // Sz ^ 2
                         &MPA_Modulus,
                         &MPA_Mod_Sub_Difference ); // result =
                                                    //     Sx - Sz ^ 2

// Step 5
// Add tempBuffer4 (Sz ^ 2) and tempBuffer1 (Sx) and
//  store in tempBuffer4.
MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer1[ 0 ];

MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer4[ 0 ];

MPA_Mod_Add_Sum.Data_Ptr = &fg_tempBuffer4[ 0 ];

(void) MPA_Mod_Add ( &MPA_Mod_Add_Augend,    // Sx
                     &MPA_Mod_Add_Addend,    // Sz ^ 2
                     &MPA_Modulus,
                     &MPA_Mod_Add_Sum );     // result = Sx + (Sz ^ 2)

// Step 6
// Multiply tempBuffer4 (Sx + (Sz ^ 2)) and tempBuffer5 (Sx - (Sz ^ 2))
//  and store in tempBuffer5.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer4[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer5[ 0 ];

(void) MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier,   // Sx + (Sz ^ 2)
                         &MPA_Mod_Mult_Multiplicand, // Sx - (Sz ^ 2)
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );    // (Sx + (Sz ^ 2)) *
                                                     //   (Sx - (Sz ^ 2))

// Step 7: NOTE: Stopped labelling content of buffers because the
//         operands were getting large

// Add tempBuffer5 to tempBuffer5 to tempBuffer5
//  and store in tempBuffer4 by performing the following steps:
MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Add_Sum.Data_Ptr = &fg_tempBuffer6[ 0 ];

(void) MPA_Mod_Add ( &MPA_Mod_Add_Augend,
                     &MPA_Mod_Add_Addend,
                     &MPA_Modulus,
                     &MPA_Mod_Add_Sum );
```

```
// Add tempBuffer5 to tempBuffer6 and store in tempBuffer4.
MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer6[ 0 ];

MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Add_Sum.Data_Ptr = &fg_tempBuffer4[ 0 ];

(void) MPA_Mod_Add ( &MPA_Mod_Add_Augend,
                     &MPA_Mod_Add_Addend,
                     &MPA_Modulus,
                     &MPA_Mod_Add_Sum );

// Step 8
// Multiply tempBuffer2 (Sy) and tempBuffer3 (Sz)
//  and store in tempBuffer3.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer2[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer3[ 0 ];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer3[ 0 ];

(void) MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier,   // Sy
                         &MPA_Mod_Mult_Multiplicand, // Sz
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );    // result = Sy * Sz

// Step 9
// Add tempBuffer3 (Sy * Sz) to tempBuffer3 (Sy * Sz)
//  and store in tempBuffer3.
MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer3[ 0 ];

MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer3[ 0 ];

MPA_Mod_Add_Sum.Data_Ptr = &fg_tempBuffer3[ 0 ];


(void) MPA_Mod_Add( &MPA_Mod_Add_Augend,   // Sy * Sz
                    &MPA_Mod_Add_Addend,   // Sy * Sz
                    &MPA_Modulus,
                    &MPA_Mod_Add_Sum );    // result = (Sy * Sz) * 2

// Step 10
// Calculate tempBuffer2 (Sy) ^ 2 and store in tempBuffer2.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer2[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer2[ 0 ];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer2[ 0 ];

(void) MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier,   // Sy
                         &MPA_Mod_Mult_Multiplicand, // Sz
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );    // result = Sy * Sz

// Step 11
// Multiply tempBuffer1 (Sx) and tempBuffer2 (Sy * Sy)
//  and store in tempBuffer5.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer1[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer2[ 0 ];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer5[ 0 ];
```

```
(void) MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier,   // Sx
                         &MPA_Mod_Mult_Multiplicand, // Sy * Sy
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );    // result =
                                                     //   Sx * (Sy * Sy)

// Step 12
// Multiply 4 * tempBuffer5 (Sx * (Sy * Sy))
//  and store in tempBuffer5 -- perform as 2 adds
//  instead of 1 multiply!
MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Add_Sum.Data_Ptr = &fg_tempBuffer6[ 0 ];

(void) MPA_Mod_Add( &MPA_Mod_Add_Augend,  //Sx * (Sy * Sy)
                    &MPA_Mod_Add_Addend,  // Sx * (Sy * Sy)
                    &MPA_Modulus,
                    &MPA_Mod_Add_Sum );   // (Sx * (Sy * Sy)) +
                                          //           (Sx * (Sy * Sy))

MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer6[ 0 ];

MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer6[ 0 ];

MPA_Mod_Add_Sum.Data_Ptr = &fg_tempBuffer5[ 0 ];

(void) MPA_Mod_Add( &MPA_Mod_Add_Augend, // see previous add
                    &MPA_Mod_Add_Addend, // see prev MPA_Mod_Add_Addend,
                                         // see previous add
                    &MPA_Modulus,
                    &MPA_Mod_Add_Sum );  // tempBuffer5 * 4

// Step 13
// Calculate tempBuffer4 (from Step 7) ^ 2
//  and save to tempBuffer1 (Sx).
// Multiply tempBuffer4 (from Step 7) by tempBuffer4 (from Step 7)
//  and save to tempBuffer1
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer4[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer4[ 0 ];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer1[ 0 ];

(void) MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier,    // from Step 7
                         &MPA_Mod_Mult_Multiplicand,  // from Step 7
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );     // tempBuffer4 *
                                                     //     tempBuffer4

// Step 14
// Subtract 2 * tempBuffer5 (from Step 12)
//  from tempBuffer1 (from Step 13) and
//  store in tempBuffer1 by performing the
//  following steps:

// Add (NOT A MODULUS ADD) temp5Buffer (from Step 12) and
//  temp5Buffer (from Step 12) and store as tempBuffer6.
MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer5[ 0 ];
```

```
        // before using temp6 again, make sure MSW is set to a 0...
        fg_tempBuffer6[ MAX_COORDINATE_SIZE_WORDS ] = 0;
        MPA_Mod_Add_Sum.Data_Ptr = &fg_tempBuffer6[ 0 ];

        (void)MPA_Add( &MPA_Mod_Add_Augend,  // from Step 12
                       &MPA_Mod_Add_Addend,  // from Step 12
                       &MPA_Mod_Add_Sum );   // result = temp5 + temp5

        // Modulus subtract tempBuffer6 (2 * tempBuffer5) from
        //  tempBuffer1 - modulus subtract will either add a p
        //  or nothing to tempBuffer1.
        MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer1[ 0 ];

        MPA_Mod_Sub_Subtrahend.Data_Length = SIZE_WORDS( fg_tempBuffer6 );
        MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer6[ 0 ];

        MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer9[ 0 ];

        (void) MPA_Mod_Subtract( &MPA_Mod_Sub_Minuend,
                                 &MPA_Mod_Sub_Subtrahend,
                                 &MPA_Modulus,
                                 &MPA_Mod_Sub_Difference ); //result =
                                                            //  tempBuffer1 -
                                                            //       tempBuffer6
//      now copy the result back into temp1
        Burst_Copy( &fg_tempBuffer1[ 0 ],          // copy to
                    &fg_tempBuffer9[ 0 ],          // copy from
                    MAX_COORDINATE_SIZE_WORDS );


        // Step 15
        // Calculate tempBuffer2 (from Step 10) ^ 2 and save to
        //  tempBuffer2 (from Step 10) by multiplying
        //  tempBuffer2 (from Step 10) by tempBuffer2 (from Step 10).
        MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer2[ 0 ];

        MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer2[ 0 ];

        MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer2[ 0 ];

        (void) MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier,    // from Step 10
                                 &MPA_Mod_Mult_Multiplicand,  // from Step 10
                                 &MPA_Modulus,
                                 &MPA_Mod_Mult_Product );     // tempBuffer2 *
                                                              //     tempBuffer2

        // Step 16
        // Multiply 8 * tempBuffer2 and store in tempBuffer2, do as 3 adds.
        // NOTE!!! on the Sierra 2 platform, it is quicker to perform
        // 1 mod mult instead of 3 Mod Adds...
        MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer2[ 0 ];

        fg_tempBuffer6[0] = 8;
        fg_tempBuffer6[1] = 0;

        MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer6[ 0 ];
        // set the length to 1
        MPA_Mod_Mult_Multiplicand.Data_Length = 1;

        MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer2[ 0 ];

        (void) MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier,    // from step 15
                                 &MPA_Mod_Mult_Multiplicand,  // * 8
```

```
                              &MPA_Modulus,
                              &MPA_Mod_Mult_Product );     // 8 * temp2

        // Step 17
        // Subtract tempBuffer1 (from Step 14) from tempBuffer5 (from Step 12)
        //  and store in tempBuffer5.
        MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer5[ 0 ];

        MPA_Mod_Sub_Subtrahend.Data_Length = MAX_COORDINATE_SIZE_WORDS;
        MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer1[ 0 ];

        MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer9[ 0 ];

        (void) MPA_Mod_Subtract( &MPA_Mod_Sub_Minuend,      // from Step 12
                                 &MPA_Mod_Sub_Subtrahend,    // from Step 14
                                 &MPA_Modulus,
                                 &MPA_Mod_Sub_Difference ); //result =
                                                            //   tempBuffer5 -
                                                            //      tempBuffer1

//      now copy the result back into temp5
        Burst_Copy( &fg_tempBuffer5[ 0 ],          // copy to
                    &fg_tempBuffer9[ 0 ],          // copy from
                    MAX_COORDINATE_SIZE_WORDS );

        // Step 18
        // Multiply tempBuffer4 (from Step 7) and tempBuffer5 (from Step 17)
        //  and store in tempBuffer5.
        MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer4[ 0 ];

        MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer5[ 0 ];
        MPA_Mod_Mult_Multiplicand.Data_Length = SIZE_WORDS( fg_tempBuffer5 );

        MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer5[ 0 ];

        (void) MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier,   // from Step 7
                                 &MPA_Mod_Mult_Multiplicand, // from Step 17
                                 &MPA_Modulus,
                                 &MPA_Mod_Mult_Product );    // tempBuffer5 *
                                                             //    tempBuffer4

        // Step 19
        // Subtract tempBuffer2 (from Step 16) from tempBuffer5 (from Step 18)
        //  and store in tempBuffer2.
        MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer5[ 0 ];

        MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer2[ 0 ];

        MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer9[ 0 ];

        (void) MPA_Mod_Subtract( &MPA_Mod_Sub_Minuend,     // from Step 18
                                 &MPA_Mod_Sub_Subtrahend,   // from Step 16
                                 &MPA_Modulus,
                                 &MPA_Mod_Sub_Difference ); //result =
                                                            //   tempBuffer5 -
                                                            //       tempBuffer2
//      now copy the result back into temp2
        Burst_Copy ( &fg_tempBuffer2[ 0 ],           // copy to
                     &fg_tempBuffer9[ 0 ],           // copy from
                     MAX_COORDINATE_SIZE_WORDS );

        // Step 20
        // Set Point_R_Ptr to Rx = tempBuffer1,
```

```
        //          Ry = tempBuffer2, & Rz = tempBuffer3.
        Burst_Copy ( Point_R_Ptr->XY_Ptr->X_Ptr,      // copy to
                     &fg_tempBuffer1[ 0 ],            // copy from
                     MAX_COORDINATE_SIZE_WORDS );

        Burst_Copy ( Point_R_Ptr->XY_Ptr->Y_Ptr,      // copy to
                     &fg_tempBuffer2[ 0 ],            // copy from
                     MAX_COORDINATE_SIZE_WORDS );

        Burst_Copy ( Point_R_Ptr->Z_Ptr,              // copy to
                     &fg_tempBuffer3[ 0 ],            // copy from
                     MAX_COORDINATE_SIZE_WORDS );
    }

    return;
}
```

```
/*****************************************************************************

FUNCTION:        EC_Mult

DESCRIPTION:
         EC_Mult accepts a projective point S, an element n and returns a
         projective point R such that R = nS.

ARGUMENTS:
    Point_S_Ptr -
            Points to the projective point to multiply by the value pointed to
            by n_Ptr, this parameter is the multiplier.

    n_Ptr -
            Points to the value to multiply the passed in projective point
            by, this is the multiplicand.

            Also contains the Number of words in the value pointed to by
            n_Ptr AND the value pointed to by Point_S_Ptr

    Modulus_Ptr -
            Points to the modulus, which is used during the Elliptic Curve
            double operation.

            Also, The length of the modulus in bytes.

    Point_R_Ptr -
            The projective point resulting, from the Elliptic Curve
            multiplication of the passed in projective point, is returned
            in this parameter.

RETURN VALUES:   The length (in words) of the data (x + y )in the projective
                 point resulting from the Elliptic Curve multiplication of the
                 passed in projective point and the value pointed to by n_Ptr.
                 Will be set to 0 if the size passed in is > max.

LIMITATIONS:     Max amount of data is 12 words.
                 R is a projective pt, and if caller wants an affine pt,
                 need to call the convert_proj_to_Affine functions.  Didn't
                 want to have to take in n prime( needed for the multiplier )

NOTES:           None

*****************************************************************************/
t_UINT32 EC_Mult( t_AFFINE_POINT*         Point_S_Ptr,
                  t_DATA_AND_SIZE_STRUCT* n_Ptr,
                  t_DATA_AND_SIZE_STRUCT* Modulus_Ptr,
                  t_PROJECTIVE_POINT*     Point_R_Ptr )
{
    t_PROJECTIVE_POINT projPointS;
    t_AFFINE_POINT     affinePointS;
    t_PROJECTIVE_DATA  S_data = { {0}, {0}, {0} };

    t_PROJECTIVE_POINT projPointR;
    t_AFFINE_POINT     affinePointR;
    t_PROJECTIVE_DATA  R_data = { {0}, {0}, {0} };

    // set nPtr to the most significant word
    t_UINT32* nPtr = n_Ptr->data_ptr + n_Ptr->data_length_words - 1;
    t_UINT32  nBit;
    t_UINT32  bitsToShift = BITS_IN_WORD;
    t_UINT32  result_data_length_words = 0;
    t_UINT32  word_count = n_Ptr->data_length_words;
```

```
// Set up the pointers
affinePointS.X_Ptr = &S_data.X_Data[0];
affinePointS.Y_Ptr = &S_data.Y_Data[0];
projPointS.XY_Ptr = &affinePointS;
projPointS.Z_Ptr = &S_data.Z_Data[0];

affinePointR.X_Ptr = &R_data.X_Data[0];
affinePointR.Y_Ptr = &R_data.Y_Data[0];
projPointR.XY_Ptr = &affinePointR;
projPointR.Z_Ptr = &R_data.Z_Data[0];


// To make sure that S data is 12 words...  copy them into a
// known buffer pre-filled with 0's.
// If there length is less than or equal to 12, than copy that data.
// otherwise do nothing, and return 0.
if( word_count <= MAX_COORDINATE_SIZE_WORDS )
{
    Burst_Copy( projPointS.XY_Ptr->X_Ptr,
                Point_S_Ptr->X_Ptr,
                n_Ptr->data_length_words );

    Burst_Copy( projPointS.XY_Ptr->Y_Ptr,
                Point_S_Ptr->Y_Ptr,
                n_Ptr->data_length_words );

    // also set the starting point of R to S, this time copy all
    // 12 words, including any trailing 0's
    Burst_Copy( projPointR.XY_Ptr->X_Ptr,
                projPointS.XY_Ptr->X_Ptr,
                MAX_COORDINATE_SIZE_WORDS );

    Burst_Copy( projPointR.XY_Ptr->Y_Ptr,
                projPointS.XY_Ptr->Y_Ptr,
                MAX_COORDINATE_SIZE_WORDS );

    // when converting an affine pt to a projective pt,
    // need to make the Z component a 1;
    R_data.Z_Data[ 0 ] = 1;
    S_data.Z_Data[ 0 ] = 1;


    // Look at the most significant bit of n
    nBit = ( ( *nPtr ) >> bitsToShift - 1 );
    bitsToShift--;

    // While the most significant bit of n is equal to 0
    while ( nBit == 0 )
    {
        // Stay here until you get a 1
        // Look at next most significant bits of n
        nBit = ( ( ( *nPtr ) >> bitsToShift - 1 ) & 1 );
        bitsToShift--;

        if ( bitsToShift == 0 )
        {
            // Need to look at next most significant byte
            bitsToShift = BITS_IN_WORD;
            word_count--;

            // Move to next most significant word
            nPtr--;
```

```
            }
        }

        // DO WHILE there are more bits in n
        do
        {
            EC_Double( &projPointR,
                       Modulus_Ptr,
                       &projPointR );

            // Look at next most significant bits of n
            nBit = ( ( ( *nPtr ) >> bitsToShift - 1 ) & 1 );
            bitsToShift--;

            if( bitsToShift == 0 )
            {
                // Need to look at next most significant byte
                bitsToShift = BITS_IN_WORD;
                word_count--;

                // Move to next most significant byte
                nPtr--;
            }

            // IF next most significant bit is a 1 THEN
            if ( nBit == 1 )
            {
                // Add R' and S and store back into R
                EC_Full_Add ( &projPointS,
                              &projPointR,
                              Modulus_Ptr,
                              &projPointR );
            }
            // Else the next bit is a 0, so do nothing but increment to next bit.
        } while ( word_count > 0 );

        // Now you have the result in projPointR, so copy back into Result buffer
        Burst_Copy( Point_R_Ptr->XY_Ptr->X_Ptr,
                    projPointR.XY_Ptr->X_Ptr,
                    n_Ptr->data_length_words );

        Burst_Copy( Point_R_Ptr->XY_Ptr->Y_Ptr,
                    projPointR.XY_Ptr->Y_Ptr,
                    n_Ptr->data_length_words );

        Burst_Copy( Point_R_Ptr->Z_Ptr,
                    projPointR.Z_Ptr,
                    n_Ptr->data_length_words );

        // The size of x + y
        result_data_length_words = n_Ptr->data_length_words * 2;
    }

    return( result_data_length_words );
}
```

```
/***************************************************************************

FUNCTION:        EC_Full_Add

DESCRIPTION:
         EC_Full_Add accepts 2 projective points S, T and performs some
         checks.  If the checks are successful, this function then calls
         the correct function and returns a projective point R such that R =
         S + T.

ARGUMENTS:
    Point_S_Ptr -
            Points to the first projective point to add.

    Point_T_Ptr -
            Points to the second projective point to add.

    Modulus_Ptr -
            Points to the modulus, which is used during the
            Elliptic Curve add operation.

    Modulus_Size_Bytes -
            The length of the modulus in bytes.

    Point_R_Ptr -
            The projective point resulting, from the addition of the two
            passed in projective points, is returned in this parameter.

RETURN VALUES:   None

LIMITATIONS:     None

NOTES:           None

***************************************************************************/
void EC_Full_Add( t_PROJECTIVE_POINT*     Point_S_Ptr,
                  t_PROJECTIVE_POINT*     Point_T_Ptr,
                  t_DATA_AND_SIZE_STRUCT* Modulus_Ptr,
                  t_PROJECTIVE_POINT*     Point_R_Ptr)
{

    // If either T or S is the point of infinity, return the opposite point.
    // The point of infinity is determined if the z coordinate is 0.
    if ( memcmp ( Point_S_Ptr->Z_Ptr,
                  &g_ZEROES[ 0 ],                  // Initialized to zero
                  MAX_COORDINATE_SIZE_BYTES ) == 0 )
    {
        // Set Point_R_Ptr to Point_T_Ptr
        //  Rx = Tx, Ry = Ty, & Rz = Tz
        Burst_Copy ( Point_R_Ptr->XY_Ptr->X_Ptr,     // copy to
                     Point_T_Ptr->XY_Ptr->X_Ptr,     // copy from
                     MAX_COORDINATE_SIZE_WORDS );

        Burst_Copy ( Point_R_Ptr->XY_Ptr->Y_Ptr,     // copy to
                     Point_T_Ptr->XY_Ptr->Y_Ptr,     // copy from
                     MAX_COORDINATE_SIZE_WORDS );

        Burst_Copy ( Point_R_Ptr->Z_Ptr,             // copy to
                     Point_T_Ptr->Z_Ptr,             // copy from
                     MAX_COORDINATE_SIZE_WORDS );
    }

    else if ( memcmp ( Point_T_Ptr->Z_Ptr,
```

```
                      &g_ZEROES[ 0 ],           // Initialized to zero
                      MAX_COORDINATE_SIZE_BYTES ) == 0 )
    {
        // Set point_R_ptr to point_S_ptr.
        //  Rx = Sx, Ry = Sy, & Rz = Sz
        Burst_Copy ( Point_R_Ptr->XY_Ptr->X_Ptr,      // copy to
                     Point_S_Ptr->XY_Ptr->X_Ptr,      // copy from
                     MAX_COORDINATE_SIZE_WORDS );

        Burst_Copy ( Point_R_Ptr->XY_Ptr->Y_Ptr,      // copy to
                     Point_S_Ptr->XY_Ptr->Y_Ptr,      // copy from
                     MAX_COORDINATE_SIZE_WORDS );

        Burst_Copy ( Point_R_Ptr->Z_Ptr,              // copy to
                     Point_S_Ptr->Z_Ptr,              // copy from
                     MAX_COORDINATE_SIZE_WORDS );
    }

    else
    {

        // Call the Elliptic Curve add function.
        _EC_Add ( Point_S_Ptr,
                  Point_T_Ptr,
                  Modulus_Ptr,
                  Point_R_Ptr );

        // If the Elliptic Curve add function returns R equal to (0,0,0)
        if ( ( memcmp ( Point_R_Ptr->XY_Ptr->X_Ptr,
                        &g_ZEROES[ 0 ],
                        MAX_COORDINATE_SIZE_BYTES ) == 0 ) &&

             ( memcmp ( Point_R_Ptr->XY_Ptr->Y_Ptr,
                        &g_ZEROES[ 0 ],
                        MAX_COORDINATE_SIZE_BYTES ) == 0 ) &&

             ( memcmp ( Point_R_Ptr->Z_Ptr,
                        &g_ZEROES[ 0 ],
                        MAX_COORDINATE_SIZE_BYTES ) == 0 ) )
        {
            // Call the Elliptic Curve double function
            EC_Double ( Point_S_Ptr,
                        Modulus_Ptr,
                        Point_R_Ptr );
        }
    }

    return;
}
```

```
/******************************************************************************

FUNCTION:       EC_Full_Sub

DESCRIPTION:
         EC_Full_Sub accepts 2 projective points S, T performs a
         subtraction then calls the add function and returns a projective
         point R such that R = S - T.

ARGUMENTS:
    Point_S_Ptr -
            Points to the first projective point.

    Point_T_Ptr -
            Points to the second projective point, which is subtracted
            from the first point.

    Modulus_Ptr -
            Points to the modulus, which is used during the
            Elliptic Curve subtraction operation.

    Modulus_Size_Bytes -
            The length of the modulus in bytes.

    Point_R_Ptr -
            The projective point resulting from subtracting the second
            passed in projective point from the first passed in projective
            point is returned in this parameter.

RETURN VALUES:   None

LIMITATIONS:     None

NOTES:           None

******************************************************************************/
void EC_Full_Sub( t_PROJECTIVE_POINT*     Point_S_Ptr,
                  t_PROJECTIVE_POINT*     Point_T_Ptr,
                  t_DATA_AND_SIZE_STRUCT* Modulus_Ptr,
                  t_PROJECTIVE_POINT*     Point_R_Ptr)
{
    t_MPA_BIG_INT_STRUCT MPA_Mod_Sub_Minuend;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Sub_Subtrahend;
    t_MPA_BIG_INT_STRUCT MPA_Modulus;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Sub_Difference;

    t_PROJECTIVE_DATA  uData;
    t_AFFINE_POINT     uPt;
    t_PROJECTIVE_POINT projUPt;


    // Initialize the pointers.
    uPt.X_Ptr = &uData.X_Data[ 0 ];
    uPt.Y_Ptr = &uData.Y_Data[ 0 ];
    projUPt.XY_Ptr = &uPt;
    projUPt.Z_Ptr = &uData.Z_Data[ 0 ];

    // Set a projective point U equal to T.
    // Only copy x and z since y will be generated.
    Burst_Copy ( projUPt.XY_Ptr->X_Ptr,               // copy to
                 Point_T_Ptr->XY_Ptr->X_Ptr,          // copy from
                 MAX_COORDINATE_SIZE_WORDS );
```

```
    Burst_Copy ( projUPt.Z_Ptr,                        // copy to
                 Point_T_Ptr->Z_Ptr,                   // copy from
                 MAX_COORDINATE_SIZE_WORDS );

    // Set Uy equal to the modulus minus Ty.
    MPA_Mod_Sub_Minuend.Data_Length = Modulus_Ptr->data_length_words;
    MPA_Mod_Sub_Minuend.Data_Ptr = Modulus_Ptr->data_ptr;

    MPA_Mod_Sub_Subtrahend.Data_Length = MAX_COORDINATE_SIZE_WORDS;
    MPA_Mod_Sub_Subtrahend.Data_Ptr = Point_T_Ptr->XY_Ptr->Y_Ptr;

    MPA_Modulus.Data_Length = Modulus_Ptr->data_length_words;
    MPA_Modulus.Data_Ptr = Modulus_Ptr->data_ptr;

    MPA_Mod_Sub_Difference.Data_Ptr = projUPt.XY_Ptr->Y_Ptr;

    (void)
     MPA_Mod_Subtract ( &MPA_Mod_Sub_Minuend,         // modulus
                        &MPA_Mod_Sub_Subtrahend,      // Ty
                        &MPA_Modulus,
                        &MPA_Mod_Sub_Difference );    // result = modulus - Ty


    // Call the Elliptic Curve full add function with S, U and R.
    EC_Full_Add( Point_S_Ptr,
                 &projUPt,
                 Modulus_Ptr,
                 Point_R_Ptr );

    return;
}
```

```
/*****************************************************************************

FUNCTION:        _EC_Add

DESCRIPTION:
         _EC_Add accepts 2 projective points S, T and returns a projective
         point R such that R = S + T.

         This function should be called by EC_Full_Add, as that function
         performs checks on the inputs that this function does not.

ARGUMENTS:
    Point_S_Ptr -
            Points to the first projective point to add.

    Point_T_Ptr -
            Points to the second projective point to add.

    Modulus_Ptr -
            Points to the modulus used during an Elliptic Curve add operation.

    Modulus_Size_Bytes -
            The length of the modulus in bytes.

    Point_R_Ptr -
            The resulting projective point, from the addition of the
            passed in projective points, is returned in this parameter.

RETURN VALUES:   None

LIMITATIONS:     None

NOTES:           None

*****************************************************************************/
void _EC_Add( t_PROJECTIVE_POINT*     Point_S_Ptr,
              t_PROJECTIVE_POINT*     Point_T_Ptr,
              t_DATA_AND_SIZE_STRUCT* Modulus_Ptr,
              t_PROJECTIVE_POINT*     Point_R_Ptr)
{
    t_MPA_BIG_INT_STRUCT MPA_Modulus;

    t_MPA_BIG_INT_STRUCT MPA_Mod_Sub_Minuend;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Sub_Subtrahend;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Sub_Difference;

    t_MPA_BIG_INT_STRUCT MPA_Mod_Add_Augend;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Add_Addend;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Add_Sum;

    t_MPA_BIG_INT_STRUCT MPA_Mod_Mult_Multiplier;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Mult_Multiplicand;
    t_MPA_BIG_INT_STRUCT MPA_Mod_Mult_Product;

    t_UINT32 i;


    Burst_Fill( &fg_tempBuffer6[0],
                &g_ZEROES[0],
                SIZE_WORDS( fg_tempBuffer6 ) );

    // Set tempBuffer1 equal to Sx (x portion of Point_S_Ptr).
    Burst_Copy ( &fg_tempBuffer1[ 0 ],                    // copy to
```

```
            Point_S_Ptr->XY_Ptr->X_Ptr,         // copy from
            MAX_COORDINATE_SIZE_WORDS );

// Set tempBuffer2 equal to Sy (y portion of Point_S_Ptr).
Burst_Copy ( &fg_tempBuffer2[ 0 ],                  // copy to
             Point_S_Ptr->XY_Ptr->Y_Ptr,        // copy from
             MAX_COORDINATE_SIZE_WORDS );

// Set tempBuffer3 equal to Sz (z portion of Point_S_Ptr).
Burst_Copy ( &fg_tempBuffer3[ 0 ],                  // copy to
             Point_S_Ptr->Z_Ptr,                // copy from
             MAX_COORDINATE_SIZE_WORDS );

// Set tempBuffer4 equal to Tx (x portion of Point_T_Ptr).
Burst_Copy ( &fg_tempBuffer4[ 0 ],                  // copy to
             Point_T_Ptr->XY_Ptr->X_Ptr,        // copy from
             MAX_COORDINATE_SIZE_WORDS );

// Set tempBuffer5 equal to Ty (y portion of Point_T_Ptr).
Burst_Copy ( &fg_tempBuffer5[ 0 ],                  // copy to
             Point_T_Ptr->XY_Ptr->Y_Ptr,        // copy from
             MAX_COORDINATE_SIZE_WORDS );


// the modulus never changes for the math functions.  Set it up once and
// never change it.
MPA_Modulus.Data_Length = Modulus_Ptr->data_length_words;
MPA_Modulus.Data_Ptr = Modulus_Ptr->data_ptr;

// The data lengths are usually = MAX_COORDINATE_SIZE_WORDS so set them
// and then leave them alone execpt for the few cases that they change.
MPA_Mod_Mult_Multiplier.Data_Length = MAX_COORDINATE_SIZE_WORDS;
MPA_Mod_Mult_Multiplicand.Data_Length = MAX_COORDINATE_SIZE_WORDS;

MPA_Mod_Sub_Minuend.Data_Length = MAX_COORDINATE_SIZE_WORDS;
MPA_Mod_Sub_Subtrahend.Data_Length = MAX_COORDINATE_SIZE_WORDS;

MPA_Mod_Add_Augend.Data_Length = MAX_COORDINATE_SIZE_WORDS;
MPA_Mod_Add_Addend.Data_Length = MAX_COORDINATE_SIZE_WORDS;

// Step1
// Set tempBuffer6 to 1 to do a compare - This is initialized to zero by
//  this function.
fg_tempBuffer6[ 0 ] = 1;

if ( memcmp  ( (t_UINT8 *)Point_T_Ptr->Z_Ptr,
               (t_UINT8 *)&fg_tempBuffer6[ 0 ],
               MAX_COORDINATE_SIZE_BYTES ) != 0 )
{
    // Set tempBuffer6 equal to Tz (z portion of point_T_ptr).
    Burst_Copy ( &fg_tempBuffer6[ 0 ],              // copy to
                 Point_T_Ptr->Z_Ptr,            // copy from
                 MAX_COORDINATE_SIZE_WORDS );

    // Set tempBuffer7 equal to tempBuffer6 (Tz) ^ 2.
    MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer6[ 0 ];

    MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer6[0];

    MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer7[0];

    (void)MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier, // Tz
                            &MPA_Mod_Mult_Multiplicand, // Tz
```

```
                            &MPA_Modulus,
                            &MPA_Mod_Mult_Product );// result = Tz ^ 2

    // Multiply tempBuffer1 (Sx) and tempBuffer7 (Tz ^ 2)
    //  and save to tempBuffer1.
    MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer1[ 0 ];

    MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer7[0];

    MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer1[0];

    (void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // Sx
                             &MPA_Mod_Mult_Multiplicand, // Tz ^ 2
                             &MPA_Modulus,
                             &MPA_Mod_Mult_Product );// result = Sx * Tz ^ 2

    // Multiply tempBuffer6 (Tz) and tempBuffer7 (Tz ^ 2)
    //  and save in tempBuffer7.
    MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer7[ 0 ];

    MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer6[0];

    MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer7[0];

    (void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // Tz ^ 2
                             &MPA_Mod_Mult_Multiplicand, // Tz
                             &MPA_Modulus,
                             &MPA_Mod_Mult_Product );// result = Tz ^ 2 * Tz

    // Multiply tempBuffer2 (Sy) and tempBuffer7 (Tz ^ 3)
    //  and save in tempBuffer2.

    MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer2[0];

    MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer2[0];

    (void) MPA_Mod_Multiply( &MPA_Mod_Mult_Multiplier, // Tz ^ 3
                             &MPA_Mod_Mult_Multiplicand, // Sy
                             &MPA_Modulus,
                             &MPA_Mod_Mult_Product );// result = Sy * Tz ^ 3

}

// Step 2
// Set tempBuffer7 (Tz ^ 3) equal to tempBuffer3 (Sz) ^ 2.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer3[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer3[0];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer7[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // Sz
                         &MPA_Mod_Mult_Multiplicand, // Sz
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );// result = Sz ^ 2

// Step 3
// Multiply tempBuffer4 (Tx) and tempBuffer7 (Sz ^ 2)
//  and save in tempBuffer4.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer7[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer4[0];
```

```
MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer4[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // Sz ^ 2
                         &MPA_Mod_Mult_Multiplicand, // Tx
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );// result = Tx * Sz ^ 2

// Step 4
// Multiply tempBuffer3 (Sz) and tempBuffer7 (Sz ^ 2)
//  and store in tempBuffer7.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer3[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer7[0];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer7[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // Sz
                         &MPA_Mod_Mult_Multiplicand, // Sz ^ 2
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );// result = Sz * Sz ^ 2

// Step 5
// Multiply tempBuffer5 (Ty) and tempBuffer7 (Sz ^ 3)
//  and store in tempBuffer5.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer5[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // Ty
                         &MPA_Mod_Mult_Multiplicand, // Sz ^ 3
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );// result = Ty * Sz ^ 3

// Step 6
// Subtract tempBuffer4 (Tx * Sz ^ 2) from tempBuffer1 (Sx * Tz ^ 2)
//  and store in tempBuffer4.
MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer1[ 0 ];

MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer4[0];

MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer9[0];

(void)MPA_Mod_Subtract( &MPA_Mod_Sub_Minuend,     // Sx * Tz ^ 2
                        &MPA_Mod_Sub_Subtrahend,  // Tx * Sz ^ 2
                        &MPA_Modulus,
                        &MPA_Mod_Sub_Difference ); // result = Sx * Tz ^ 2 -
                                                   //          Tx * Sz ^ 2

// Now copy the result back into tempBuffer4.
Burst_Copy( &fg_tempBuffer4[ 0 ],          // copy to
            &fg_tempBuffer9[ 0 ],          // copy from
            MAX_COORDINATE_SIZE_WORDS );

// Step 7
// Subtract tempBuffer5 (Ty * Sz ^ 3) from tempBuffer2 (Sy * Tz ^ 3)
//  and store in tempBuffer5.
MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer2[ 0 ];

MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer5[0];

MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer9[0];

(void)
```

```
   MPA_Mod_Subtract ( &MPA_Mod_Sub_Minuend,     // Ty * Sz ^ 3
                      &MPA_Mod_Sub_Subtrahend,  // Sy * Tz ^ 3
                      &MPA_Modulus,
                      &MPA_Mod_Sub_Difference ); //Ty * Sz ^ 3 - Sy * Tz ^ 3

// Now copy the result back into tempBuffer5.
Burst_Copy ( &fg_tempBuffer5[ 0 ],                    // copy to
             &fg_tempBuffer9[ 0 ],                    // copy from
             MAX_COORDINATE_SIZE_WORDS );

// If contents of tempBuffer4 (from Step 6) is NOT equal to 0
if ( memcmp ( &fg_tempBuffer4[ 0 ],
              &g_ZEROES[0],
              MAX_COORDINATE_SIZE_BYTES ) != 0 )
{
    // Step 9
    // Subtract tempBuffer4 (from Step 6) from
    //  2 * tempBuffer1 (Sx * Tz ^ 2) and store in tempBuffer1
    //  by performing the following steps:


    // Add (NOT A MODULUS ADD) tempBuffer1 (Sx * Tz ^ 2) and
    //  tempBuffer1 (Sx * Tz ^ 2) and store as tempbuffer8.
    MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer1[ 0 ];

    MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer1[ 0 ];

    MPA_Mod_Add_Sum.Data_Ptr = &fg_tempBuffer8[ 0 ];

    (void)MPA_Add( &MPA_Mod_Add_Augend,// Sx * Tz ^ 2
                   &MPA_Mod_Add_Addend,//  Sx * Tz ^ 2
                   &MPA_Mod_Add_Sum );// Sx * Tz ^ 2 + Sx * Tz ^ 2


    // Modulus subtract tempBuffer4 (from Step 6) from
    //  tempBuffer8 (2 * (Sx * Tz ^ 2)) and store in tempBuffer1.
    MPA_Mod_Sub_Minuend.Data_Length = SIZE_WORDS( fg_tempBuffer8 );
    MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer8[ 0 ];

    MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer4[0];

    MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer1[0];

    (void)  MPA_Mod_Subtract( &MPA_Mod_Sub_Minuend,     // Ty * Sz ^ 3
                              &MPA_Mod_Sub_Subtrahend,  // Sy * Tz ^ 3
                              &MPA_Modulus,
                              &MPA_Mod_Sub_Difference ); // 2 * (Sx * Tz ^ 2)
                                                      //       from Step 6

    // Step 10
    // Subtract tempBuffer5 (from Step 7) from
    //  2 * tempBuffer2 (Sy * Tz ^ 3) and store in
    //  tempBuffer2 by performing the following steps:


    // Add (NOT A MODULUS ADD) tempBuffer2 (Sy * Tz ^ 3) and
    //  tempBuffer2 (Sy * Tz ^ 3) and store in tempBuffer8.
    MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer2[ 0 ];

    MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer2[ 0 ];

    // make sure MSW of temp8 is set to 0
    fg_tempBuffer8[ MAX_COORDINATE_SIZE_WORDS ] = 0;
```

```
(void)MPA_Add( &MPA_Mod_Add_Augend,// Sy * Tz ^ 3
               &MPA_Mod_Add_Addend,// Sy * Tz ^ 3
               &MPA_Mod_Add_Sum );// Sy * Tz ^ 3 + Sy * Tz ^ 3


// Modulus subtract tempBuffer5 (from Step 7) from
//  tempBuffer8 (from Step 10) and store in tempBuffer2.
MPA_Mod_Sub_Minuend.Data_Length = SIZE_WORDS( fg_tempBuffer8 );

MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer5[0];

MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer2[0];

(void)MPA_Mod_Subtract( &MPA_Mod_Sub_Minuend,     // from Step 10
                        &MPA_Mod_Sub_Subtrahend,  // from Step 7
                        &MPA_Modulus,
                        &MPA_Mod_Sub_Difference ); //r = temp8 - temp5

// Set tempBuffer8 to 1 to do a compare.
Burst_Fill ( &fg_tempBuffer8[ 0 ],
             &g_ZEROES[0],
             SIZE_WORDS( fg_tempBuffer8 ) );

fg_tempBuffer8[ 0 ] = 1;

// If Tz does not equal 1
if ( memcmp ( Point_T_Ptr->Z_Ptr,
              &fg_tempBuffer8[ 0 ],
              MAX_COORDINATE_SIZE_BYTES ) != 0 )
{
    // Multiply tempBuffer3 (Sz) and tempBuffer6 (Tz)
    // and store in tempBuffer3.
    MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer3[ 0 ];

    MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer6[0];

    MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer3[0];

    (void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // Sz
                             &MPA_Mod_Mult_Multiplicand, // Tz
                             &MPA_Modulus,
                             &MPA_Mod_Mult_Product );// result = Sz * Tz

}

// Step 11
// Multiply tempBuffer3 (Sz * Tz) and tempBuffer4 (from Step 6)
//  and store in tempBuffer3.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer3[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer4[0];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer3[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // Sz * Tz
                         &MPA_Mod_Mult_Multiplicand, // from Step 6
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );// result = (Sz * Tz) *
                                                 // from Step 6

// Step 12
// Multiply tempBuffer4 (from Step 6) and tempBuffer4 (from Step 6)
```

```
//  and store in tempBuffer7.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer4[ 0 ];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer7[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // from Step 6
                         &MPA_Mod_Mult_Multiplicand, // from Step 6
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );// result = from Step 6
                                                 // from Step 6

// Step 13
// Multiply tempBuffer4 (from Step 6) and tempbuffer7 (from Step 12)
//  and store in tempBuffer4.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer7[ 0 ];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer4[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // from Step 6
                         &MPA_Mod_Mult_Multiplicand, // from Step 12
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );// result = from Step 6
                                                 //   * from Step 12

// Step 14
// Multiply tempBuffer1 (from Step 9) and tempBuffer7 (from Step 12)
//  and store in tempBuffer7.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer1[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer7[0];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer7[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // from Step 9
                         &MPA_Mod_Mult_Multiplicand, // from Step 12
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );// result = from Step 9
                                                 //   * from Step 12

// Step 15
// Multiply tempBuffer5 (from Step 7) and tempBuffer5 (from Step 7)
//  and store in tempBuffer1.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer5[0];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer1[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // from Step 7
                         &MPA_Mod_Mult_Multiplicand, // from Step 7
                         &MPA_Modulus,
                         &MPA_Mod_Mult_Product );// result = from Step 7
                                                 //   * from Step 7

// Step 16
// Subtract tempBuffer7 (from Step 14) from tempBuffer1 (from Step 15)
//  and store in tempBuffer1.
MPA_Mod_Sub_Minuend.Data_Length = MAX_COORDINATE_SIZE_WORDS;
MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer1[ 0 ];

MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer7[0];

MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer9[0];
```

```
(void)
   MPA_Mod_Subtract ( &MPA_Mod_Sub_Minuend,     // from Step 15
                      &MPA_Mod_Sub_Subtrahend,  // from Step 14
                      &MPA_Modulus,
                      &MPA_Mod_Sub_Difference );// result = from Step 15
                                                //  -   from Step 14

// Now copy the result back into tempBuffer1.
Burst_Copy( &fg_tempBuffer1[ 0 ],
            &fg_tempBuffer9[ 0 ],
            MAX_COORDINATE_SIZE_WORDS );


// Step 17
// Subtract 2 * tempBuffer1 (from Step 16) from
//  tempBuffer7 (from Step 14) and store in tempBuffer7
//  by performing the following steps:

// Add (NOT A MODULUS ADD) tempBuffer1 and tempBuffer1
//  and store in tempBuffer8.
MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer1[ 0 ];

MPA_Mod_Add_Addend.Data_Ptr = &fg_tempBuffer1[ 0 ];

// make sure MSW of temp8 is set to 0
fg_tempBuffer8[ MAX_COORDINATE_SIZE_WORDS ] = 0;

(void)MPA_Add( &MPA_Mod_Add_Augend,// from Step 16
                   &MPA_Mod_Add_Addend,// from Step 16
                   &MPA_Mod_Add_Sum );// from Step 16 + from Step 16


 // Modulus subtract tempBuffer8 (from Step 17) from
 //  tempBuffer7 (from step 14)
 MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer7[ 0 ];

 MPA_Mod_Sub_Subtrahend.Data_Length = SIZE_WORDS( fg_tempBuffer8 );
 MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer8[0];

 MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer9[0];

 (void)
    MPA_Mod_Subtract ( &MPA_Mod_Sub_Minuend,     // from Step 14
                       &MPA_Mod_Sub_Subtrahend,  // from Step 17
                       &MPA_Modulus,
                       &MPA_Mod_Sub_Difference );//  from Step 14
                                                 //  -   from Step 17

 // Now copy the result back into tempBuffer7.
 Burst_Copy ( &fg_tempBuffer7[ 0 ],
              &fg_tempBuffer9[ 0 ],
              MAX_COORDINATE_SIZE_WORDS );

// Step 18
// Multiply tempBuffer5 (from Step 7) and tempBuffer7 (from Step 17)
//  and store in tempBuffer5.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer7[ 0 ];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer5[0];

(void)MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // from Step 17
                         &MPA_Mod_Mult_Multiplicand, // from Step 7
```

```
                             &MPA_Modulus,
                             &MPA_Mod_Mult_Product );// result = from Step17
                                                     //   * from Step 7

// Step 19
// Multiply tempBuffer2 (from Step 10) and tempBuffer4 (from Step 13)
//  and store in tempBuffer4.
MPA_Mod_Mult_Multiplier.Data_Ptr = &fg_tempBuffer2[ 0 ];

MPA_Mod_Mult_Multiplicand.Data_Ptr = &fg_tempBuffer4[0];

MPA_Mod_Mult_Product.Data_Ptr = &fg_tempBuffer4[0];

(void) MPA_Mod_Multiply ( &MPA_Mod_Mult_Multiplier, // from Step 10
                             &MPA_Mod_Mult_Multiplicand, // from Step 13
                             &MPA_Modulus,
                             &MPA_Mod_Mult_Product );// result = from Step10
                                                     //   * from Step 13

// Step 20
// Modulus subtract tempBuffer4 (from Step 19) from
//  tempBuffer5 (from Step 18) and store in tempBuffer2.
MPA_Mod_Sub_Minuend.Data_Ptr = &fg_tempBuffer5[ 0 ];

MPA_Mod_Sub_Subtrahend.Data_Length = MAX_COORDINATE_SIZE_WORDS;
MPA_Mod_Sub_Subtrahend.Data_Ptr = &fg_tempBuffer4[0];

// make sure MSW of temp8 is set to 0 before using
fg_tempBuffer8[ MAX_COORDINATE_SIZE_WORDS ] = 0;
MPA_Mod_Sub_Difference.Data_Ptr = &fg_tempBuffer8[0];

(void)
   MPA_Mod_Subtract ( &MPA_Mod_Sub_Minuend,     // from Step 18
                      &MPA_Mod_Sub_Subtrahend,  // from Step 19
                      &MPA_Modulus,
                      &MPA_Mod_Sub_Difference );//  from Step 18
                                                //  -  from Step 19

// Step 21
// Divide tempBuffer8 (from Step 20) by 2 by performing the
//  following steps:

// If contents of tempBuffer2 (from Step 20) are odd
if ( ( fg_tempBuffer8[ 0 ] & 1 ) == 1 )
{
    // Add p to tempBuffer2 and store in tempBuffer2.
    MPA_Mod_Add_Augend.Data_Ptr = &fg_tempBuffer8[ 0 ];

    MPA_Mod_Add_Addend.Data_Ptr = Modulus_Ptr->data_ptr;

    (void)MPA_Add( &MPA_Mod_Add_Augend,// from Step 20
                   &MPA_Mod_Add_Addend,// p
                   &MPA_Mod_Add_Sum );// result = from Step 20 + p
}

// Right shift tempBuffer8 by 1 bit and store in tempBuffer2.
for( i = 0; i <= (MAX_COORDINATE_SIZE_WORDS - 1); i++ )
{
    fg_tempBuffer2[ i ] = ( ( fg_tempBuffer8[ i ] >> 1 ) |
                            ( fg_tempBuffer8[ i + 1 ] << 31 ) );
}

// Step 22
```

```
    //  Set Point_R_Ptr to
    //    Rx = tempBuffer1, Ry = tempBuffer2, and Rz = tempBuffer3.
    Burst_Copy ( Point_R_Ptr->XY_Ptr->X_Ptr,      // copy to
                 &fg_tempBuffer1[ 0 ],               // copy from
                 MAX_COORDINATE_SIZE_WORDS );

    Burst_Copy ( Point_R_Ptr->XY_Ptr->Y_Ptr,      // copy to
                 &fg_tempBuffer2[ 0 ],               // copy from
                 MAX_COORDINATE_SIZE_WORDS );

    Burst_Copy ( Point_R_Ptr->Z_Ptr,             // copy to
                 &fg_tempBuffer3[ 0 ],              // copy from
                 MAX_COORDINATE_SIZE_WORDS );
}

// Step 8 continued

else // Contents of tempBuffer4 (from Step 6) are equal to 0.
{
    // If tempBuffer5 (from Step 7) is equal to 0
    if ( memcmp ( &fg_tempBuffer5[ 0 ],
                  &g_ZEROES[0],
                  MAX_COORDINATE_SIZE_BYTES ) == 0 )
    {
        // Set Point_R_Ptr to
        //  Rx = 0, Ry = 0, and Rz = 0.
        Burst_Fill ( Point_R_Ptr->XY_Ptr->X_Ptr,
                     &g_ZEROES[0],
                     MAX_COORDINATE_SIZE_WORDS  );

        Burst_Fill ( Point_R_Ptr->XY_Ptr->Y_Ptr,
                     &g_ZEROES[0],
                     MAX_COORDINATE_SIZE_WORDS );

        Burst_Fill ( Point_R_Ptr->Z_Ptr,
                     &g_ZEROES[0],
                     MAX_COORDINATE_SIZE_WORDS );
    }

    else // Contents of tempBuffer5 (from Step 7) are NOT equal to 0.
    {
        // Set Point_R_Ptr to
        //  Rx = 1, Ry = 1, and Rz = 0.

        // Set Rx to all zeroes.
        Burst_Fill ( Point_R_Ptr->XY_Ptr->X_Ptr,
                     &g_ZEROES[0],
                     MAX_COORDINATE_SIZE_WORDS );

        // Set Rx to 1.
        *(Point_R_Ptr->XY_Ptr->X_Ptr) = 1;

        // Set Ry to all zeroes.
        Burst_Fill ( Point_R_Ptr->XY_Ptr->Y_Ptr,
                     &g_ZEROES[0],
                     MAX_COORDINATE_SIZE_WORDS );

        // Set Ry to 1.
        *(Point_R_Ptr->XY_Ptr->Y_Ptr) = 1;

        // Set Rz to all zeroes.
        Burst_Fill ( Point_R_Ptr->Z_Ptr,
                     &g_ZEROES[0],
```

```
                                    MAX_COORDINATE_SIZE_WORDS );
            }
        }

        return;
    }
```