Executive Summary

Certain applications pose tremendous challenge for their implementation on resource constrained embedded systems. More often than not, the reason is intensive computations in the algorithms used for their implementation. So while designing custom accelerators usually an application is considered and efforts are spent on optimizing its execution. This usually is a hardware-software partition problem where the designer offloads certain tasks of the application algorithm from host processor to parallel capabilities of hardware. The designer has to tradeoff between execution time gain , area increase and power consumption.

High performance systems can be built if above approach is extended to multiple applications. Demanding computing tasks of multiple applications can be identified to check for similarities and accelerated by hardware. Thus gains could be seen over multiple applications with minimum overheads. But it is not easy to do this and generic Instruction set extension (ISE) is an effort in this direction. We wanted to evaluate this same strategy, but over the very popular ARM Instruction set architecture. We wished to check if we can accelerate multiple applications on ARM systems by proposing instruction set extensions beneficial to most applications with minimum overheads. To do this it was necessary to leverage ARM's existing capabilities and propose well researched extensions to it. Also, it was decided undertake this effort over an industrial platform like Cortex M1, rather than a simulator so that any benefits could be taken advantage of right away.

We chose Elliptic curve cryptography (ECC) and Fast Fourier transform for this approach and identified computation tasks in them. We were able to identify certain operations common to them but already very well supported by ARM. Application algorithms were intensely researched and other instruction set extensions proposed were studied. The analysis led to the decision of implementing MULGF2 extension. This was because this capability could be introduced in the existing hardware with minimum overhead. The unit used to perform this action is configurable to act as a general purpose integer multiplier or multiply over binary finite fields. The hardware peripheral was implemented and performance was measured on our platform. Efficient algorithms to target our new hardware peripheral were also used. Thus what we achieved was -

- Efficient implementation of MULGF2 extension along with comparison of ECC performance using 3 different binary field multiplication algorithms(section 5.2)
- Ascertain costs of adding MULGF2 to existing ARM platform (area and power)
- Decide efficient strategies for instruction set extension in soft core systems
- Best ECC implementation figures (to the best of authors knowledge) compared to similar soft-core embedded platforms (see Table 6)

Acknowledgement

I would like to express my sincere gratitude for the help and guidance from my supervisor Dr. Simon Hollis. I thank him the most for always being patient with my constant queries.

I would be grateful to my co-supervisor Dr. Jose Nunez-Yanez for providing all the resources needed for this project and introducing me to this exciting technology.

I would also like to thank the library staff especially for the 'How to' lectures which broadened my vision beyond Google.

I am grateful to my friend and fellow classmate Pranav for helping me at a crucial point of my project.

Lastly, I would thank my family and friends for their persistent support and encouragement.

LIST OF ABBREVIATIONS

AHB	AMBA High performance Bus
ALU	Arithmetic logic unit
AMBA	Advanced Microcontroller Bus Architecture
ANSI	American National Standards Institute
APB	Advanced Peripheral Bus
ASIC	Application-specific integrated circuit
CRC	cyclic redundancy check
DSA	Digital Signature Algorithm
DSP	Digital signal processing/processor
ECC	Elliptic curve cryptography
EEMBC	Embedded Microprocessor Benchmark Consortium
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
FPGA	Field-programmable gate array
GPIO	General purpose input output
ISA	Instruction set architecture
ISE	Instruction set extension
JPEG	Joint Photographic Experts Group
LUT	Look up table
NIST	National Institute of Standards and technology
TI OMAP	Texas instruments Open Multimedia Application Platform
RSA	Rivest, Shamir and Adleman
SECG	Standards for Efficient Cryptography Group

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Today embedded systems have to support a range of applications and provide a variety of functionalities to the end user. We have systems like Smart phones, tablets and single board computers which work in GHz range while the other end of the spectrum shows devices like smart sensors and actuators working in the range of few MHz. But what is common is the ever increasing pressure to support new capabilities, protocols and technology. The challenge in the case of latter is to work with very less resources. The chip real estate and the device performance in watts rather milliwatts is a concern.

Some applications demand intensive computations due to the nature of the algorithms used in them, for instance algorithms for signal processing, multimedia, floating point arithmetic, cryptography etc. To support multiple applications with good or even reasonable performance on a small embedded system is a challenging task and there have been various approaches towards it. The most common approach to address computation intensive applications has been to design specialized hardware accelerators for them. This is widely used in FPGA and ASIC platforms where a central processor is supported by custom designed hardware accelerators. Accelerators designed in such a way are not general purpose and lead to a heterogeneous system with increased development time and sometimes unacceptable area and power overhead.

Instruction-set extension is another approach which can provide reasonable gains. Complex SIMD machines with specialized instructions for high performance exist. These coprocessors are generally built with special purpose hardware design for e.g. to support Image processing and the very recent 3D technology [1]. These coprocessors with their instructions prove extremely capable in their environments. But we wish to investigate instruction set extensions for cost-sensitive pervasive computing devices. We are talking about extensions which go easy on hardware resources, flexible in nature and general purpose.

Such hardware extensions to a processor would need efficient hardware-software co-design because some tasks run well in software. This would also mean considering the features (instructions) of the underlying embedded platform as certain type of operations are well supported. This is primarily because general purpose processor and microcontroller designers have over the course of time started providing increasing support in the form of instructions. Typical examples include hardware floating point support, bit manipulation instructions; specialized digital signal processing (DSP) instruction set extension and so on. In particular, we are interested in investigating the ARM instruction set. The proliferation of ARM processors into platforms and devices of all sizes is very well known and it can be safely said that everyone owns at least one ARM based device. It has an evolving instruction set to support its wide usage and popular for designing optimum

performance/cost systems. We are again reiterating our vision to make our hardware extension as general purpose as possible i.e. useful over a wide variety of applications.

A soft-core processor is a processor in RTL format and present in the bit-stream of a FPGA [2]. It has its own instruction set like any other microprocessor or a microcontroller. Typically, soft cores cover a small part of the available FPGA fabric and the rest can be utilized to develop hardware designs. The advantage of a soft core based embedded system is the ability to utilize sequential control and management capabilities of a processor and parallel nature of hardware. The soft core and the hardware peripherals are connected by a bus structure or protocol. Several proprietary soft-cores like Cortex M1 from ARM [3], Leon3 from Gaisler research, Nios from Altera, MicroBlaze from Xilinx [4] are available for designs now. The traditional advantages of a FPGA which include flexibility and very less NRE (Non recurring Engineering) costs are well known. In addition to the above, these soft core systems help us build small compact systems by incorporating a processor into the design and easy migration to an ASIC of the same. While the concern is higher power dissipation to performance ratio compared to an ASIC's or even normal FPGA designs, recent developments like Flash based FPGA's with sleep modes and extreme fast wake up times address this to an extent [5]. These soft-core processors provide us with an ideal platform to evaluate our hardware-software co-design scenario. The flexibility for execution of a task or operation over a soft-core processor (sequential instructions) or to accelerate by parallel hardware is available. We use industry standard ARM's soft-core Cortex M1 for our design.

Having said this, practice of proposing hardware extensions is not new but they tend to remain as a part of research environment. It takes long for them to appear in industry standard platforms if not never and become available for common use. Some soft-core systems like the ones based on NIOS compensate this by allowing us to define instructions using processor ALU data path. Costs incurred by transfer of operands over on chip bus can be avoided. But a platform like Cortex M1 won't allow this tightly coupled configuration and the performance of such systems on instruction set extension needs to be evaluated. Thus to investigate performance of instruction set extension in a loosely coupled configuration is also an objective.

As a summary we can state the **aims and objectives** of our project in the following way:

AIM: Research efficient Instruction set extension (ISE) to ARM ISA in a multi-application scenario and evaluation over ARM Cortex M1 soft-core

To achieve this, the objectives to be achieved are;

1. Comprehensive study of typical applications with algorithms to implement them

We want to investigate whether in multi-application scenarios we could identify computation intensive tasks inherent to majority of them and offload them to hardware. The number of applications studied is subject to time constraints.

2. Study of existing ARM ISA features

The suitability of the application algorithms under consideration could be measured by the degree to which ARM architecture would favour it, thus demonstrating a rather unusual architecture-centric approach. As already mentioned, developments have taken place since ARM ISE first appeared in the digital scene to support modern application scenarios.

3. Effective hardware-software partitioning for ISE

The computation intensive tasks which appear common in different algorithms could be potential candidates for ISE i.e. for implementation in Hardware. This can be understood by implementing algorithms in SW and profiling to find demanding tasks. Gains on implementation of the peripheral and accessed as a part of Instruction set (software) could be predicted.

4. To implement hardware peripheral and modify software to access the hardware module

5. Measurement of performance gain, code size change, area and power overheads of ISE to Cortex M1

The structure of the thesis is outlined as below. The next chapter details the chosen applications for our approach and their theoretical concepts. It tells us in brief about ARM ISA and similar works done in the past. Chapter 3 lists all the analysis that went into design of the hardware peripheral we chose for ARM ISE. It is one of the key chapters which explains in detail our approach towards design, decisions taken and projected gains. Chapter 4 lists out the hardware and software development carried out along with brief description of various tools and technologies used in the process. Chapter 5 is a collection of observations and results of the work carried out along with their analysis. Here we list the actual gains obtained, the overheads occurred in the process and give more information about the hardware we developed. Chapter 6 is a retrospect of our aims and objectives along with a critical evaluation of our choices, methodology and implementation. Chapter 7 outlines work which can be further done as a continuation to this thesis.

Chapter 2 Background

This chapter explains various concepts necessary to understand the work carried out. We present the basics of Elliptic curve cryptography and different parameters associated with its implementation. We also explain fast Fourier transform (FFT) and the radix-2 algorithm we chose to work with. The next section 'ARM Instruction set architecture' lists salient features of the ISA relevant to our thesis. In the last section 'Previous work' we do a comprehensive review of work and approaches similar to our thesis.

2.1 Applications

The application which primarily interested us was cryptography. In this digital age, huge amounts of data are being transmitted and received. Secure communication now is the need of not only military and government organisations but also individuals. This raises a concern over privacy and also authenticity of data received by the user. This necessity has given rise to the field of advanced cryptography and it has been intensely researched for the past few decades. Present age applications such as digital cash, smart cards, e-commerce imply significance of efficient cryptography. Also important is the reliability of the underlying systems which carry out this important task. Public key cryptography schemes like RSA and Elliptic curve cryptography (ECC) are preferred over symmetric key schemes for information exchange as no private keys are transmitted over communication channels. A party wishing to communicate will take the receiver's public key which is freely visible to all and encode the message with it. The message will be decoded by the receiver with the help of a private key unique to it. The public-private key pair shares a mathematical relation which allows the sender to encrypt message using public key while the receiver can decrypt it using the public key. Common Cryptography techniques based on RSA (Rivest, Shamir and Adleman). or DSA (Digital signature algorithm) involve a lot of modular exponentiation operations like

$$a^b \bmod n.$$

The operands **a** and **b** are typically of bit sizes 1024 to 2048. Thus they are very resource intensive for even 32 bit MCU(microcontroller unit) platforms. On the flip side, ECC provides comparable and at times more security assurance level with operands of sizes 160 to 256 bits. As clearly shown in [6] on a 8 bit platform MCU, 160 bit ECC algorithms outperform 1024 bit RSA (Rivest, Shamir and Adleman). ECC stands out from the others in terms of considerably lesser operand sizes but they utilize an additional level of algorithms over modular arithmetic operations. An exhaustive study to justify use of ECC

on constrained devices demonstrating speed ups and less memory requirements can be seen in [6].

In accordance with our approach, we considered a second application i.e Fast Fourier Transform (FFT). This was done to consider FFT implementation algorithm and investigate if we could find fine grained similarities with ECC implementation techniques. If they turn out computationally intensive they can serve as potential candidates for hardware acceleration. There exist a lot of applications in which a signal is measured by a sensor and some processing has to be done to extract information from them. In most cases, it is transformed into frequency domain to make a note of its spectral properties. A fourier transform of the signal has to be done in this case. The methods to this are DFT – discrete Fourier Transform and FFT –Fast Fourier Transform. FFT is preferred as its complexity is $O(N \log_2 N)$ instead of $O(N^2)$ for DFT where N is the number of samples of the signal represented as a N-dimensional vector. In systems like Wireless sensor networks, IOT (Internet of things), small analog monitoring units form a part of the big system where they communicate with each other and may contain confidential data [7]. Other instances include highly classified military surveillance systems, space exploration systems and biomedical instruments. In all these applications, both efficient ECC and FFT algorithms may be needed. Another important thing is that if ECC in considered over binary finite fields (explained later), the optimisations for operations over binary field operations will also prove helpful for error-control coding techniques as shown in [8]. Also, FFT is the technique used for various other operations like DCT (Discrete Cosine Transform) and Inverse DCT for JPEG. In [9], the author lists algorithms for ECC where FFT can aid for efficient implementation. Thus it makes perfect sense to consider FFT and ECC together although at first glance they appear to be completely different in context.

2.1.1 Elliptic curve Cryptography (ECC) *[10]*

In 1985, Miller [11] and Koblitz [12] independently proposed using elliptic curves for building secure public cryptosystems. Although the topic of ECC has been researched for more than two decades now, it is a relatively new scheme to be implemented and accepted by organizations like ANSI(American National Standards Institute), NIST(National Institute of Standards and technology) and IEEE.

In this section we explain basics of Elliptic curve cryptography and the underlying finite field arithmetic. Implementation details have also been discussed. A finite field is a finite set of numbers with special properties and ECC aims at exploiting behavior of elliptic curves over these numbers. Considerable amount of parameters are involved in ECC implementation like selection of a particular curve, point representation co-ordinates, underlying finite field etc. We present details only relevant for this thesis and a

comprehensive guide can be found in [10]. We use [10] as a reference throughout this section.

An elliptic curve over Galois field $\mathbb{F}_{2^m}$ (Binary fields), can be defined by the following simplified Weierstrass equation:

$$\mathbf{E}/\mathbb{F}_{2^m} \quad : y^2 + xy = x^3 + ax^2 + b \text{.....(1) [10]}$$

Any solution $h\,(x, y)\ \epsilon\ \mathrm{F}_{2^m}$ to the above equation then lies on that elliptic curve and is called as point on the curve.

More information about finite fields is presented later.

All the ECC schemes are based on a single group operation called as **Point multiplication** which is basically adding a point P to itself $(k-1)$ times to get another point Q on the curve

$$Q = k \cdot P \text{ [10]}$$

The group is the set of all points on the elliptic curve along with a point at infinity $\mathcal{O}$. The addition required for above calculation is done by means of arithmetic operations over underlying finite field $\mathbb{F}$ described in detail in the following sections. To calculate k when P and Q are given, is not possible by any sub-exponential-time algorithm as noted in [13] and popularly called as the Elliptic curve discrete logarithm problem (ECDLP). The operand size of P and Q in bits is almost ten times less than other widely used cryptography schemes lie RSA which makes it an efficient scheme to be implemented for secure embedded systems communication [6].

The following figure can be used to represent ECC implementation structure.

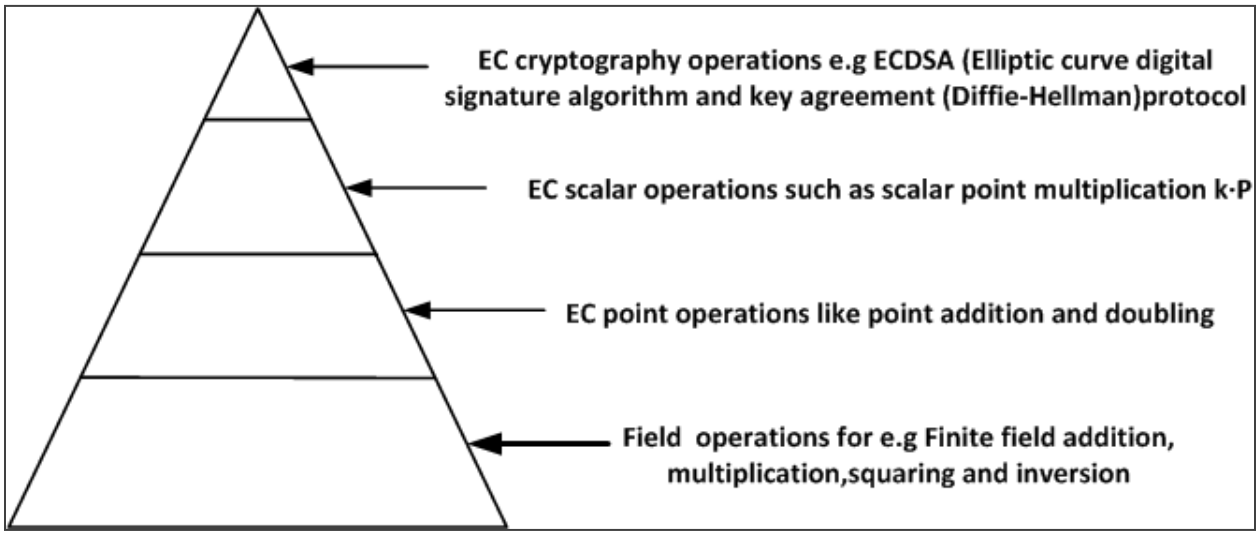


Figure 1 ECC Implementation hierarchy

Figure 1 shows layers in a typical ECC implementation. At the lowest level lies the field operations of the finite field the elliptic curve has been chosen over. Section 2.1.1.1 describes such operations of a binary finite field. The second level consists of EC point operations that allow operations over points on the curve. This is because the elliptic curve has been defined over a finite field and any operations on points of the curve are ultimately field operations. Examples include point addition i.e. Addition of two points on an elliptic curve and point doubling. The use of such EC point operations enables to perform a higher level scalar operation such as multiplication of a EC point by a scalar value. Scalar multiplication is the core operation in most of the EC cryptography operations and protocols like ECDSA and key exchange protocols. These cryptography operations fulfill the final objectives of encryption, authentication and signature verification [14]. Let us take a detailed view of each level.

2.1.1.1 Finite fields

All elliptic curve group operations require finite field operations like addition, multiplication, squaring and subtraction. A finite field is a finite set F of elements with addition (+) and multiplication (·) operations while the elements satisfy certain arithmetic operations. The number of elements in the field is called as the order of the field. Multiplication in the set is associative i.e

$$a \cdot b = b \cdot a \quad \forall\, a, b \in \mathbb{F}$$

It contains an identity element $1 \in \mathbb{F}$ and

for every $a \neq 0, a \in F$ there exists an element $a^{-1} \in \mathbb{F}$ such that $aa^{-1} = a^{-1}a = 1$.

An example of finite field would be $\mathbb{F}_7$ also called as Galois field GF(7) which consists of all positive integers between 0 and 6. They are also called as prime finite fields and 7 is the prime number in the above case. All the operations over this group are modulo 7 which means result of multiplication of 5 and 6 is

$$5 * 6 \ (mod\ 7) = 2$$

Similarly other operations can be done by reducing the result modulo 7. In this thesis, we concentrate on one type of finite field called as the binary fields or characteristic-two finite fields $\mathbb{F}_{2^m}$ or GF(2^m). The operations over these fields involve bit manipulation and more commonly implemented on hardware. So efficient hardware-software systems have been researched in this field and this interests us [15] [16] [14] [17]. The elements of the field are binary polynomials with coefficients 0 or 1 and the degree of the polynomial is less than m. For e.g. $\mathbb{F}_{2^4}$ is a set of all binary polynomials of the form

$$p_3z^3 + p_2z^2 + p_1z + p_0 \quad \text{where} \quad p_i \in \{0,1\}$$

Arithmetic here is also modulo over a chosen polynomial of degree m which is 4 in this case. Consider the reduction polynomial

$$f(z) = (z^4 + z + 1)$$

Therefore, multiplication over this field of two elements $(z^3 + 1)$ and $(z^2 + 1)$ would be

$$(z^3 + 1)(z^2 + 1)\, mod\ (z^4 + z + 1) = (z^3 - z + 1)$$

Other operations along with algorithms to execute them are described below.

We construct binary field elements using polynomial basis representation as it is appropriate for hardware or software implementation. It can be shown as

$$\mathbb{F}_{2^m} = \{ a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \ldots\ldots\ldots + a_2z^2 + a_1z + a_0 : a \in \{0,1\} \}$$

$$\approx \{ (a_{m-1}a_{m-2} \ldots\ldots a_1a_0) : a \in \{0,1\} \} \Rightarrow \text{Binary string of length m}$$

An irreducible polynomial $f(z)$ of degree m i.e. a polynomial which cannot be factored as a product of binary polynomials each of degree *m-1* or less is chosen as reduction polynomial. The result of field multiplication needs to be reduced similar to modulo of numbers to keep the field finite (degree less than m). The additive identity is string of all 0's while the multiplicative identity is (000…0001). Addition of two elements is just bitwise XOR of the coefficients (because it is modulo 2).

i.e if $p = (p_{m-1}p_{m-2} \ldots\ldots p_1p_0)$ and $q = (q_{m-1}q_{m-2} \ldots\ldots q_1q_0)$ then

$$c = p + q = (c_{m-1}c_{m-2} \ldots\ldots c_1c_0) \text{ Where } c_i = p_i\ XOR\ q_i$$

Also,

$p \cdot q = r = (r_{m-1}r_{m-2} \ldots\ldots r_1r_0)$ where r is the remainder polynomial when

$(p_{m-1}z^{m-1} + p_{m-2}z^{m-2} + \ldots.. + p_2z^2 + p_1z + p_0) \cdot (q_{m-1}z^{m-1} + q_{m-2}z^{m-2} + \ldots.. + q_2z^2 + q_1z + q_0)$

is divided by chosen irreducible polynomial $f(z)$ of degree m.

This section gives description of the various algorithms used for binary finite field operations like addition and multiplication. As already noted, ECC point operations are done using these underlying field operations. The algorithms depend on the data bus width of the underlying processor. We denote this parameter by width W. It can be 8, 16, 32 or even 64. Each polynomial consisting of $m - 1$ coefficients can be written as a bit stream of length m

$$\text{i.e } a(z) = (a_{m-1}a_{m-2} \ldots\ldots a_1a_0)$$

No conventional processor can support operand of sizes 163, 233, 283 etc which typically are degree size of operands (value of m) in ECC. This makes it necessary to break the operand bits into blocks of size typically W. So we have $t = \lceil m/W \rceil$. The operands are represented in array notation as

$$a[z] = \{ a[0], a[1] \ldots \ldots a[t-2], a[t-1] \} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(2)$$

and they can be manipulated to operate on the long binary arithmetic operands. The most significant word $a[t-1]$ is padded with zeroes if necessary.

In these thesis, we implement binary field operations over field $\mathbb{F}_{2^{163}}$. The advantage of using this field is that it is one of those recommended by NIST and SECG (Standards for Efficient Cryptography Group). The ARM Cortex M1 is a 32 bit soft-core processor and $W = 32$. Therefore,

$$t = \lceil 163/32 \rceil = 6$$

in our case . The operand is represented as an array of 6 unsigned integer data type. The upper 29 bits of $a[t-1]$ $i.e$ $a[5]$ are filled with zeroes. The algorithms given for the operations have been directly adapted from [10] and we acknowledge the importance of this resource in the thesis.

2.1.1.1.1 Addition

Addition over field elements is performed by bitwise logical XOR. Thus, it needs t word operations. The algorithm is simple XOR of coefficients of polynomials.

Algorithm 1 Addition **(see Algorithm 2.32** [10] **, page 47)**

Input: Binary polynomials $p(z)$ and $q(z)$ of maximum degree $m-1$

Output: $r(z) = p(z) + q(z)$

1. for j from 0 to $t-1$ **do**

$r[j] \leftarrow p[z] \oplus q[z]$

end for

2.1.1.1.2 Multiplication

Multiplication is one of the most demanding operation and also the focal point of investigation in the thesis. The most naïve method performing it is the shift-and-add method which is described next.

Now,

$p(z)\cdot q(z) = p_{m-1}z^{m-1}\ q(z) + p_{m-2}z^{m-2}\ q(z) + \ \ldots\ldots\ + \ p_2 z^2\ q(z) + \ p_2 z^2\ q(z) + p_0 q(z)$

In the following algorithm in each iteration i, $z^i\ q(z)\ mod\ f(z)$ is performed and added to accumulator a if $p_{i-1} = 1$.

$$\text{Now, } q(z) \odot (z) = q_{m-1}z^m + \ q_{m-2}z^{m-1} + \ \ldots\ldots + q_2 z^3 + q_1 z \ \ + \ q_0\, z$$

If $q(z)\ mod\ f(z)$ is denoted by $r(z)$ then above equation turns to,

$$q(z) \odot (z) = q_{m-1}\ r(z) \ \ + (q_{m-2}z^{m-1} + \ \ldots\ldots + q_2 z^3 + q_1 z \ \ + \ q_0\, z)\ (mod\ f(z))$$

It follows that $q(z) \odot (z)\ mod\ f(z)$ is left shift of $q(z)$ and added to $r(z)$ if MSB q_{m-1} is 1.

Algorithm 2 Multiplication by Shift-and-add **(see Algorithm 2.33** [10] **, page 48)**

Input: Binary polynomials $p(z)$ and $q(z)$ of maximum degree $m-1$

Output: $r(z) \ = \ p(z) \ \cdot \ q(z)\ mod\ f(z)$

1. **If** $p_0 = 1$ **then** $r \leftarrow q$

   else $r \leftarrow 0$

   end if

2. **for** i from 1 to $m-1$ **do**

   $q \leftarrow q \odot z\ mod\ f(z)$

   If $p_i = 1$ **then** $r \leftarrow r + q$

   end for

But as mentioned in [10], Right-to-left comb method could be used for optimizing execution because if $q(z)z^k$ has been calculated $q(z)z^{Wn+k}$ can be calculated by appending n zeroes to the right of $q(z)z^k$. If we use A notation $r(n)$ for a truncated array from n to t *-1* i.e

$$r(n) = \{\, a[n], a[n+1] \ldots \ldots a[t-2], a[t-1]\}$$

then the algorithm can be shown as follows.

Algorithm 3 Right to Left Comb method for Polynomial multiplication **(see Algorithm 2.34** [10] **, page 49)**

Input: Binary polynomials $p(z)$ and $q(z)$ of maximum degree $m-1$

Output : $r(z) = p(z) \cdot q(z) \; mod \; f(z)$

1. $r(z) = 0$
2. **for** k from 0 to $W-1$ **do**
   - $u \leftarrow 0$
   - **for** j from 0 to $t-1$ **do**
     - **if** k^{th} bit of $p[j] = 1$ **then**
       - Add $q(z)$ to $r(j)$
     - **end if**
   - **end for**
   - **if** $k \neq W-1$ **then**
     - $q(z) \leftarrow q(z) \odot z$
   - **end if**

end for

2.1.1.1.3 Squaring

For a polynomial $p(z)$, squaring is done by just inserting a zero bit between consecutive bits of *p(z)* . Thus squaring would result in

$$p(z)^2 = (0, p_{m-1}, 0, p_{m-2} \ldots \ldots 0, p_1, 0, p_0)$$ (see section 2.3.4 [10] , page 52)

2.1.1.1.4 Reduction

As already shown above, if the result of multiplication and squaring of binary polynomials in field $\mathbb{F}_{2^m}$ is more than m, the result is reduced by a polynomial of degree m similar to modulo. Such polynomials $r(z)$ can have maximum degree of $(m-1) + (m-1)$ $i.e$ $2m-2$. So, in our case with m = 163, the maximum degree could be 324 and we would need an array of 11 unsigned integers to represent it. The best approach in a resource-scarce embedded system is to implement over a known field like $\mathbb{F}_{2^{163}}$ because

well researched and documented optimizations for their implementations exist. For e.g. a fast reduction algorithm using the below reduction polynomial can be used.

$$f(z) = (z^{163} + z^7 + z^6 + z^3 + 1) \text{ .....(3)}$$

Algorithm 4 Fast reduction modulo $f(z) = (z^{163} + z^7 + z^6 + z^3 + 1)$ ... (with $W = 32$)
(see Algorithm 2.41 [10] **, page 55)**

Input: Binary polynomial r(z) and q(z) of maximum degree $2m - 2 = 324$

Output: $r(z)\ mod\ f(z)$ of degree 163

1. **for** i from 10 down to 6 **do**

   $u \leftarrow r[i]$

   $r[i-6] \leftarrow r[i-6] \oplus (u \ll 29)$

   $r[i-5] \leftarrow r[i-5] \oplus (u \ll 4) \oplus (u \ll 3) \oplus u \oplus (u \gg 3)$

   $r[i-5] \leftarrow r[i-4] \oplus (u \gg 28) \oplus (u \gg 29)$

   end for

2. $u \leftarrow r[5] \gg 3$
3. $r[0] \leftarrow r[0] \oplus (u \ll 7) \oplus (u \ll 6) \oplus (u \ll 3) \oplus u$
4. $r[1] \leftarrow r[1] \oplus (u \gg 25) \oplus (u \gg 26)$
5. $r[5] \leftarrow r[5]\ \&\ 0{\times}07$

The array $r[6] = \{r[5], r[4], r[4], r[3], r[2], r[1], r[0]\}$ contains the reduced polynomial in this case.

2.1.1.1.5 Inversion

Inversion is the most computationally intensive binary field operation. Different techniques to evaluate like Almost Inversion method, Fermat's little theorem exist and can be found in (see section 2.3.6 [10] , page 57). They have not been implemented in this thesis for reasons highlighted in the next section.

2.1.1.2 EC point operations

The most important point operation for elliptic curve points would be point addition because it is used for point multiplication. It involves addition of a point $P(x_1,y_1)$ to another curve point $Q(x_2,y_2)$ or itself.

Let $P(x_1, y_1)$ and $Q(x_2, y_2) \in \mathbb{F}_{2^m}$. So there exists a point $A(x_3, y_3)$ such that

$$P + Q = (x_3, y_3) \text{ where}$$

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad \text{.....(4)}$$

$$y_3 = \lambda (x_1 + x_3) + x_3 + y_1 \quad \text{.....(5)}$$

where $\lambda = (y_1 + y_2)(x_1 + x_2)^{-1}$ when P ≠ Q i.e point addition

$= x_1 + y_1 (x_1)^{-1}$ when P = Q i.e point doubling [14]

The above equation is valid for points represented in affine coordinates. We can see that for each addition or doubling a field division and a field multiplication is required. Division is performed by inversion and proves very costly. We use Lopez-Dahab projective coordinates [18] in which an elliptic point is represented by a tuple (X, Y, Z) instead of (x, y) standard affine representation. The relation between them can be represented as

$$(X, Y, Z) \Rightarrow (X/Z, Y/Z^2).$$

In this case the elliptic curve equation (1), changes to

$$Y^2 + XYZ = X^3 Z + a X^2 Z^2 + b XZ^4$$

Use of projective coordinates increase the number of multiplications for point addition and doubling but decrease number of divisions or inversion. Only one inversion is required at the end to convert coordinates back to standard affine representation. More information about projective coordinates and corresponding equations to calculate point addition and doubling can be found in [18]. Studies in [19] clearly show that for a variety of curves, projective coordinates are the most appropriate option for implementing ECC in low power systems. Mixed coordinate systems are also used where only one of the operands for addition or doubling is represented in projective coordinates and the result is stored in projective coordinates. The choice of the above coordinates is also justified because our efforts were aimed at accelerating binary field multiplication and these coordinates are suitable for systems with comparatively faster field multiplication than inversion.

2.1.1.3 EC scalar operations

EC point operations when performed a definite number of times result in EC scalar point operations. A multiplication of an integer with a point on the curve is nothing but adding the point to itself number of times the magnitude of the integer. For e.g. $Q = k \cdot P$ can be represented as

$$Q = P + P + P + \ldots\ldots P \ (k\ times) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5)$$

or, it can be represented as series of doubling and add as follows for e.g.

$$\text{If } \ k = 3\,, Q = \ 2P \ + 1$$

$$\text{If } \ k = 7, \ Q = \ 2(2P + 1) \ + 1$$

and so on. This reduces number of operations required. The addition of points over a curve is cyclic i.e adding a point to itself n number of times where n represents number of points on the curve will result in the same point. Thus on an elliptic curve $Q = n \cdot P = P$ where n is number of points on the curve. This fundamental property of an elliptic curve is exploited for cryptography.

Similar to [16], using lemma 4 from [18], the following table shows the amount of computations required for the corresponding operations.

ECC point operations	Number of field additions	Number of field Multiplications	Number of field Squarings	Number of field Inversions
Point Addition	2	4	1	0
Point Doubling	1	2	4	0
Converting affine to Projective coordinates	1	0	2	0
Converting Projective to affine coordinates	6	10	1	1
Scalar Multiplication $Q = k \cdot P$	$3\lfloor log_2 \mathrm{k} \rfloor + 7$	$6\lfloor log_2 \mathrm{k} \rfloor + 10$	$5\lfloor log_2 \mathrm{k} \rfloor + 3$	1

Table 1 Operations using Lopez-Dahab Projective coordinates

Point multiplication forms the basis of all cryptography operations and they also depend on certain other parameters for e.g. selection of the elliptic curve, selection of irreducible polynomial, the appropriate protocol for key exchange and signature. In small embedded systems these parameters are fixed and follow the much recommended curves by NIST or SECG [10]. The advantage of using these curves is that several optimisation techniques have been developed and documented for them. One example is the reduction polynomial

for field $\mathbb{F}_{2^{163}}$ as already discussed. A comprehensive list of standards can be found in APPENDIX B of [10].

2.1.2 Fast Fourier Transform

The motivation to study Fast Fourier transform (FFT) or rather its implementation algorithm was to find similarities with ECC implementation algorithms in context to computation intensive tasks.

A sampled signal is usually not just a sine or cosine wave but rather by Fourier's theory a weighted sum of these waves. Any waveform in time domain can be represented as a sum of individual sine and cosine waves of different frequencies. Frequency components of the wave will have their own amplitude and phases. A Fourier transform converts a continuous signal in time domain to frequency domain but we would need Discrete Fourier Transform (DFT) in case of discretely sampled signals.

A Fourier transform is given by the following equation

$$X(f) = F\{x(t)\} = \int_{-\infty}^{\infty} x(t)\ e^{-j2\Pi ft}\ dt \quad \text{.....(6)}$$

Thus, for a signal represented discretely in time the discrete Fourier transform or DFT is given by

$$X_k = \sum_{i=0}^{N-1} x_i\, e^{-j2\Pi ik/N} \qquad \text{for } k = 0,1,2\ to\ N-1 \quad \text{.....(7)}$$

Here x is the input sequence, X is the DFT and n is the number of samples in both time and frequency domain. Thus we can see the above equation that calculation of DFT involves (N * N) complex multiplications and thus its complexity is O(N^2).

In [20], the authors proposed Cooley-Tukey Fast Fourier transform technique. FFT is advantageous because it splits the data set which is N in this case into small blocks which are then further split into smaller blocks. This is done by utilizing the cyclic property of exponential in the DFT expression. It is possible to calculate DFT of N points by combining DFT of two N/2 points. The DFT of those N/2 points is also possible by similarly breaking it into two halves and so on. This FFT algorithm that works on 'divide and conquer' strategy is also called as radix-2 algorithm. It begins by calculating $N/2$ 2-point DFT's and there would be $\log_2 N$ stages to get the final N point DFT. The complexity of this algorithm would be (N * $\log_2$N) which makes a considerable difference when N is large such as 128 , 256 etc.

If we use $W_N = e^{-j2\Pi/N}$ then we have the DFT equation as

$$X_k = \sum_{i=0}^{N-1} x_i\, W_N^{ik} \qquad \text{for } k = 0,1,2\ to\ N-1 \quad \text{.....(8)}$$

If we break the above samples into even and odd parts we could write it as in [21]

$$X_k = \sum_{i=0}^{N-1} x_i W_N^{ik} = \sum_{r=0}^{N/2-1} x_{2r} W_N^{(2r)k} + \sum_{r=0}^{N/2-1} x_{2r+1} W_N^{(2r+1)k}$$

$$= \sum_{r=0}^{N/2-1} x_{2r} W_{N/2}^{(r)k} + W_N^k \left(\sum_{r=0}^{N/2-1} x_{2r+1} W_{N/2}^{(r)k}\right) \text{.....(9)}$$

$$= A_k + W_N^k B_k$$

because $W_N^{(2r)} = W_{N/2}^{(r)}$

A_k and B_k are in fact , $N/2$ point DFT's of even and odd samples in the original samples respectively. A_k and B_k can be again recursively divided such that ultimately we reach 2-point DFT's amongst samples.

From equation (9) for a 2 point DFT , we would have the following equation

$$X_k = x_0 W_2^{(0)k} + x_1 W_2^{(1)k} \quad \text{for } k = 0,1$$

and can be graphically represented as the following butterfly diagram.

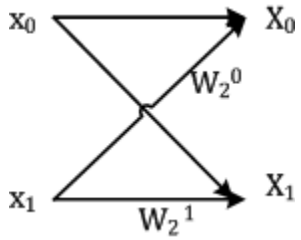


Figure 2 Butterfly graph

Using 'divide and conquer' strategy a 8 point FFT can be shown as in the **Figure 3** which is actually breaking 8 point into two 4-point FFT of even and odd elements and then further into four 2-point FFT's. Also by the formula,

$$W_N^{(2r)} = W_{N/2}^{(r)}$$

we can express all the multipliers as powers of same W_N .

What is happening is that we are avoiding a large number of calculations by not calculating each component of DFT. Instead, at each stage we start with N samples and do a 'butterfly' on them get N samples for next stage.

Also, to avoid memory fetches after each stage we could use an in-place FFT algorithm using a single data buffer of N complex numbers. That is, at the end of each stage the same data buffer is acted upon. This is also called as in-place execution. Thus the whole

system would take input in time domain and at the end of operation the same buffer would contain samples in frequency domain. It is necessary to re-order input samples for this. The strategy used is called as bit reversal of input samples [22] [21].

Consider we start with 8 samples. So the indices of the input samples would be 0, 1, 2 to 7 and they can be represented in binary notation as 000, 001, 010 to 111. To order input samples for in place FFT the bits have to be reversed i. e

000	000	0
001	100	4
010	010	2
011	110	6
100	001	1
101	101	5
110	011	3
111	111	7

Thus the samples would be ordered as x_0 , x_4 , x_2 , x_6 , x_1 , x_5 , x_3 and x_7.

The following flow diagram can be used to understand 8-point FFT. Thus we can calculate frequency components for any N using the same concept as shown below.

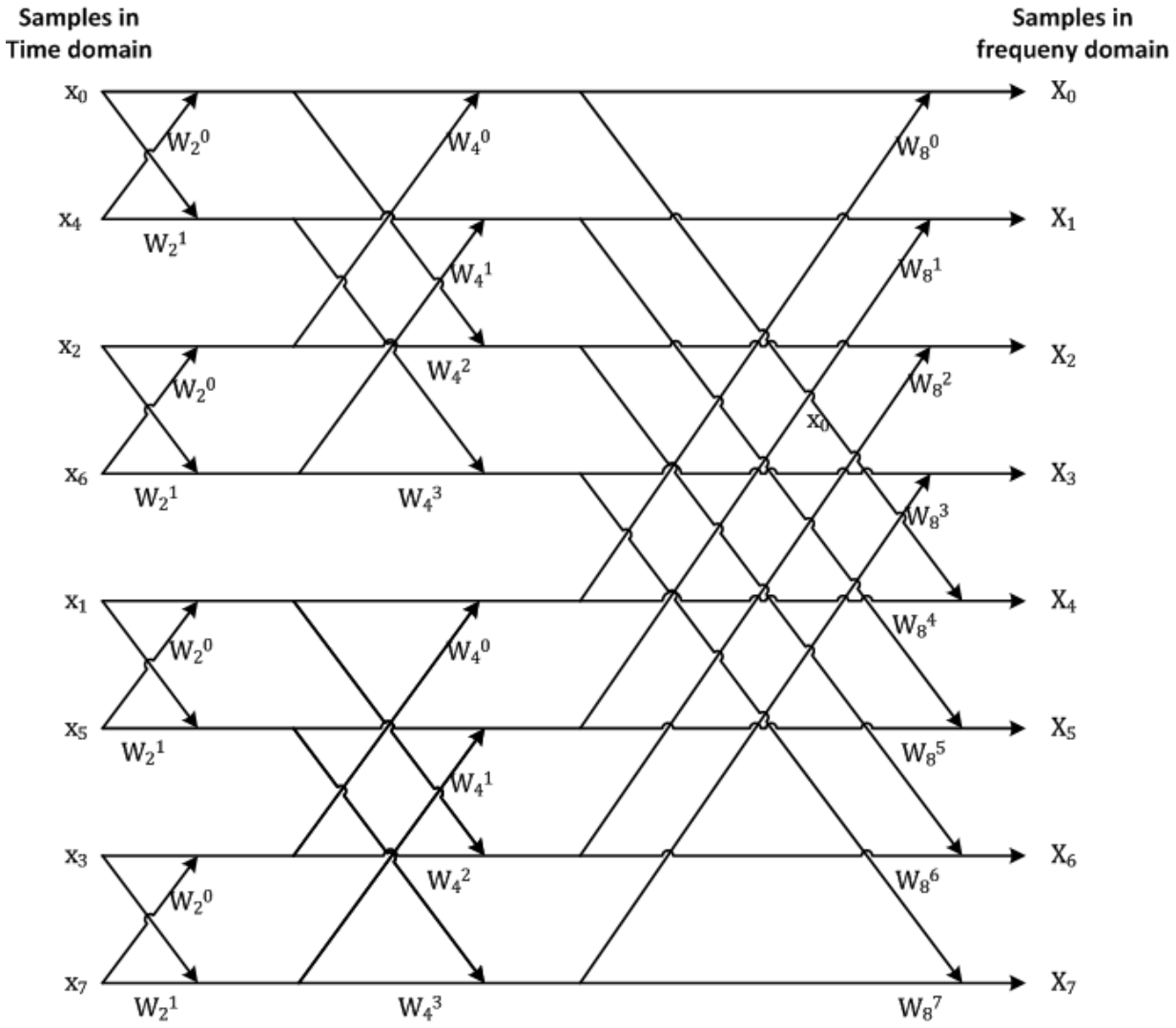


Figure 3 8-point FFT calculation

So in summary, the steps to perform decimation in time radix-2 FFT are [21]

1. Find number of stages required by calculating $\log_2 N$

2. Samples are collected in a buffer of size N.

3. Using bit-reverse technique, sort samples for in place FFT.

4. Pre-compute values of W_N required at each stage.

5. Apply butterfly to adjacent samples in the buffer

6. Now, apply butterfly to samples separated by a distance of 2.

7. Now, apply butterfly to samples separated by a distance of 4.

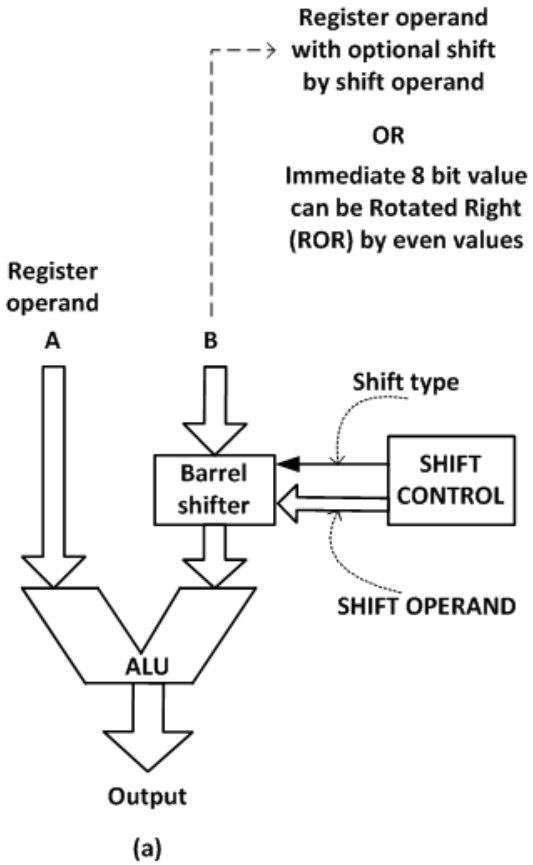
8. Continue till a distance of N/2 is reached and the buffer now has **N point FFT.**

2.2 ARM Instruction set architecture (ARM ISA)

ARM is the most widely used 32 bit RISC ISA in the field of embedded systems. This is true in terms of number of 'ARM-powered' units produced and used. The main reason for this accelerated growth in the last decade is the ability to design and develop high performance, low-cost and low-power devices with the help of ARM processor technology. Less hardware complexity enables to have small die size and reduced power consumption [2]. We wanted to evaluate the design for multi-application strategy on an ARM based system. Soft-core processors are an ideal evaluation scenario for our approach as we have a processor, an efficient bus-communication strategy and FPGA canvas to implement custom hardware accelerators. ARM provides Cortex M1 soft-core which is an ARM core specifically designed for an FPGA. So we can execute efficient algorithms on ARM soft-core and use AMBA bus protocol [23], to design hardware accelerators for Cortex M1 soft-core.

ARM supports standard RISC features of uniform register file, simple register addressing modes and a Load-Store architecture, all register-based allowing no direct access to memory. The instructions are of size 32 bits and aligned with 4 byte boundaries. It features a link register, stack pointer, program counter and a CPSR (current program status register) in USER mode of execution. In addition there exists stack pointer and SPSR (saved program status register) in PRIVILEGED mode of execution. Also, all ARM instructions can be made to behave as a NOP (no operation) or normal instruction with the help of condition flags set in PSR. These help replace small chunks of 'if-else' or 'while' code with branch-less instructions thus improving performance. A comprehensive list of other features of ARM ISA can be found in [2] and [24] . We focus on the following features of instruction set which would enable us to write efficient code for the above chosen applications. The features of ARM ISA have been analyzed before proposing extensions to it.

Data processing instructions use two source operands and one destination register. Out of the two source operands one can be register, a shifted register or an immediate value. The second operand can be made to rotate, shift logically or arithmetically with the help of inline **Barrel shifter** [2] (Refer **Figure 4 (a)**). The values for these shifts are given by an immediate or a fourth register. These shifts are carried out before the second operand is passed to the Arithmetic Logic Unit (ALU). If the shift value is specified as immediate then the shift is free while it may take an extra cycle if register is used. The use of these shifts along with other operations like ADD, SUB, RSUB (reverse subtract) etc is that they allow us to generate constants (done by compiler mostly) and also simplify intensive operations like multiply by a constant.



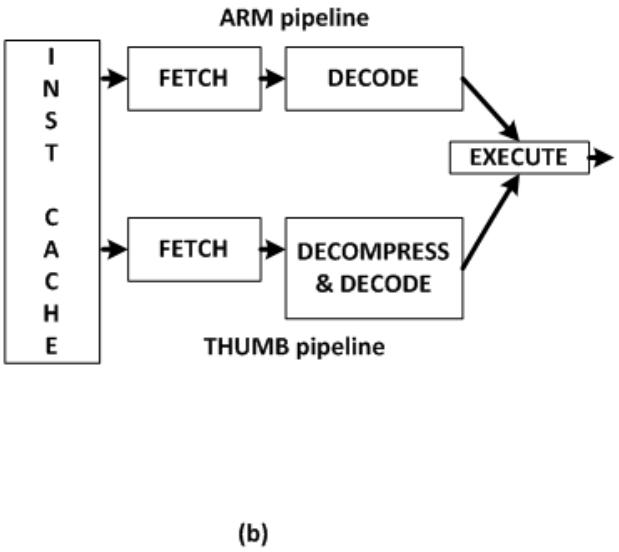


Figure 4 (a) ARM inline barrel shifter **(b) ARM and THUMB pipeline block diagram**

A simple example is to multiply a value by a constant which is of the form $(2^n \pm 1)$. Consider $r3 = r4 * 9$ which can be $r3 = r4 + r4 * 8$ and can be compiled as,

$$ADD\ r3, r4, r4, LSL\ \#3$$

for a single cycle execution. This is because a Left or right shift of a number by n is nothing but its multiplication or division respectively by2^n. In another type of instruction supported out of 12 bits available to store second operand, 8 are used to store a number between 0 – 255 and they can be ROR'ed by even numbers (0,2,4…30 which is twice of number stored in rest 4 bits) to give us a number of immediate constants. They can generate numbers bigger than what a simple 12 bit representation would do.

The shift instructions are listed as

- LSL (Logical shift left)
- LSR (logical shift right)
- ASR (Arithmetic shift right – Right shift with sign bit preservation for 2's complement operation)
- ROR (Rotate right –LSB wraps around MSB)
- RRX (Rotate right extended – Rotate right with carry acting as 33^{rd} bit).

It also provides with logical operations such as EOR (logical XOR) , AND , ORR (logical OR) and bit clear (BIC).

ARM provides both Multiply (MUL) and Multiply-accumulate (MLA) for 32 bit operands generating a 64 bit result. Also we could use the following instructions for a 64 bit result (in two registers)

- UMULL (unsigned multiply long)
- UMLAL (unsigned multiply-accumulate long)
- SMULL (signed multiply long)
- SMLAL (signed multiply-accumulate long)

The combination of ALU and shifts will allow us to do fast bit manipulations required by binary field operations we chose for ECC. Also logical operations like EOR and BIC would help because although not specified we assume these instructions to be single cycle. The general multiply instruction would be highly used for multiplication with twiddle factors and also for binary field operations in some cases. The performance of FFT would highly depend upon performance of underlying multiplier.

A unique instruction CLZ (count leading zeros) is provided in architecture v5T [2] which can help us know actual bit size of the operand. It can be used for normalizing in case of floating point operations (floating point multiplications in case of FFT). Another instruction provided with architectures like Cortex M3 is the RBIT instruction. This instruction reverses the bits of the operand in a single cycle which is much required for implementing fast bit reversals in case of FFT.

Also, an ARM innovation is a 16 bit instruction set called the THUMB which can be used for high code densities in cost sensitive applications. The code density improves by 30-35 % [25] which makes it better than 8-16 bit CISC/RISC controllers. It contains a subset of the ARM ISA where 32 bit instructions are compressed into 16 bit opcodes and later decompressed dynamically in ARM instruction pipeline [2]. They are executed as normal 32 bit ARM instructions on the same micro-architecture and thus performance is high (Refer Figure 4 (b). It lacks capability to make all instructions work with condition flags and thus contains more branches than normal ARM code. It still has smaller size of instruction memory. THUMB and ARM states are interchangeable dynamically which allows designer to have control over both performance and code density. An improvement over this was the THUMB-2 instruction set which allows all 16 bit and some 32 bit ARM instructions like BL (Branch with link), MRS (store PSR to a register), MSR (restore PSR back from register) etc to be executed without change of state overheads. This allows for further optimum battery performance in small embedded systems.

2.3 Earlier Work

2.3.1 Multi-application scenarios

Very few efforts have been directed towards effective multi-application design and some examples are [26] and [27]. In [27], the author investigates arithmetic operation accelerators for soft-core processors. The typical applications considered in [27] include Motion estimation kernels, signal processing kernels like Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters and Binary coded decimal(BCD). The author demonstrates that using hardware 'addition based building blocks' it is possible to create accelerators for the above applications with a primary aim of reusing hardware resources. The approach is to build parameterized hardware module which can perform all tasks like SAD (Sum of Absolute differences), matrix by vector multiplications, fixed point multiply, fixed point multiply-accumulate and BCD arithmetic which are exactly the software performance limiting factors in the above mentioned applications.

In [26], the authors demonstrate an approach where they investigate multiple applications and identify similar subtasks in applications at a coarse level e.g DCT which occurs in many video processing algorithms. They call them as nodes and assign metrics to them depending on their deadline, how frequently they are used, their criticality and performance/area trade-off. Based on these metrics, they create a co-design problem and propose algorithms for their efficient implementation on embedded systems. We wish to evaluate tasks at a further fine grained level because what we aim is to propose Instruction set extensions to ARM architecture. The hardware accelerators we design should give us acceleration over applications but should not be a significant area and delay overhead. In [28], the authors evaluate performance of a Floating point unit [FPU] added as a peripheral to cortex M1 over AMBA bus and can be used for varied number of applications.

2.3.2 Our applications

Next we look at different strategies used to accelerate the applications we have chosen. Hardware acceleration using FPGA's due to their flexibility and highly parallel nature is a norm. Authors in [29] using FPGA showed speed up of 150 for ECC over software implementation. In [30], authors designed a complete FFT coprocessor to perform 1024 point FFT's in microseconds. Application algorithms can be accelerated using their mathematical properties. We get a comprehensive list of such algorithms for ECC in [10]. Also, algorithms like radix-4 and mixed radix [31] can be used for considerable gains over radix-2 FFT algorithms. But we are interested in optimizations that show a considerable amount of hardware software co design approach and use of instruction set extensions. Not surprisingly, the above works concentrate on a single application domain.

2.3.2.1 DSP instructions

Exploiting VLIW (Very large instruction word) architecture using DSP's for ECC acceleration offers limited gains as described below. In (see page 55 Table II [32]) authors demonstrate how instruction XORMPY present in TI C64x+ VLIW DSP along with parallel DSP capabilities can be used to speed up binary multiplication and even out-perform 500 MHz ARM . This function can be used to perform carry-less polynomial multiplication which is same as binary field multiplication. Then they make use of the above fact to speed up ECC on a existing industry platform TI's OMAP. TI's OMAP is a heterogeneous platform consisting of ARM cores, DSP and some other utility peripherals. It is found that, calls to DSP core for performing underlying binary arithmetic had about 150 *us* communication overhead. As we have already seen from **Table 1**, number of such operations is required for one point multiplication. This forced them to perform complete 'crypto-operations' like **Point multiplication** on DSP for reasonable gains.

As seen from [33], traditionally DSP's have been preferred for FFT's and similar signal processing operations. The VLIW architecture allows operations on more data samples at same time. Multiple ALU's aid this structure. Efficient instructions for signal processing which allow loading multiple signal samples and manipulating them exist. As demonstrated by [34], FFT algorithms can be highly optimized by DSP assembly instructions.

2.3.2.2 Hardware Software co-design approach

This approach is more prevalent for ECC than FFT due to the wide number of platforms it is implemented on. Two ECC point multiplication implementations in a soft-core platform have been detailed here. In [35] , authors use Pico Blaze provided by Xilinx while in [17] authors use NIOS soft-core provided by Altera.

In [35], the authors build a scalable ECC system with Pico Blaze soft-core. The central idea is to build a scalable point multiplication and reduction accelerator in hardware and with the help of software use the system for a set of curves. Here the main CPU i.e. Pico blaze directly controls the arithmetic accelerators data path. It also controls direct data transfer of operands and partial results between memory and the units. Thus the limited bus width of Pico Blaze does not result into any hindrances. But in this approach, the entire high computation task has been offloaded while Pico Blaze only provides control signals.

The next work [17] shows a system where a feature of adding custom instructions provided by NIOS soft-core is utilized. The designer can design custom hardware blocks, combine with the processor's ALU and access by a software Macro. The hardware block if combinational can execute in a single cycle thus allowing single cycle execution for

operations which take up multiple cycles in terms of default processor instructions. Also number of clock cycles can be specified for sequential blocks. The data is moved in and out into the accelerator unit by the processor in chunks of 32 bits. After moving in the whole data, the custom logic is scheduled by custom instruction to carry out 163 bit point multiplication. In addition they add a Divider custom logic as a slave over Avalon bus interface. This combination allows them to achieve ECC operations in as fast as $0.75\,us$ for $50\,MHz$ clock frequency. This system would be useful in stand-alone ECC systems as a considerable amount of area has been used to design the whole 163 bit multiplier in hardware.

Both the above works can be shown by the following diagram. This is justified as both Nios and Pico Blaze are small processors with no inbuilt HW support for shift and XOR. It can be seen that, although SHIFT and XOR are general purpose HW units they are interleaved within binary arithmetic blocks making them difficult to use for other purposes. Also, the boundary has moved more towards software making the hardware cover major part of computations required for ECC point multiplication.

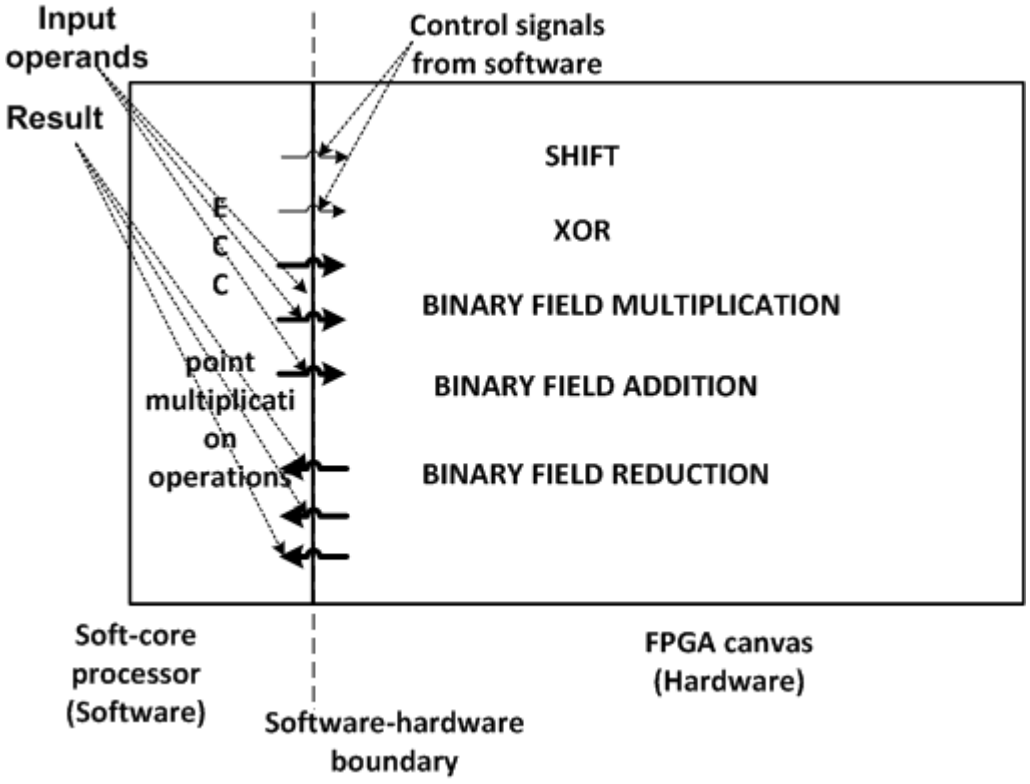


Figure 5 Software-hardware boundary for ECC implementation in earlier works

FFTW [36] is an approach used in systems which need to run FFT operations repeatedly. In a one-time effort, a 'planner' analyses different algorithms that can be run in an optimized way on a particular hardware. It is a heuristic method and can break FFT tasks recursively into simple structures that can be run in a straight line optimized code. This approach is quite complex and intensive for small embedded systems.

2.3.2.3 Other Instruction set extensions

In [37], a scalable dual field multiplier was demonstrated which did both integer and polynomial multiplication. This could be used over prime finite fields and binary finite fields. In [38], the same above multiplier in size W (data bus width of processor) was used to provide for a custom instruction MULGF2 which is multiplication over Galois field (2^m). Along with this they designed word level algorithms for binary field arithmetic. These algorithms are similar to what we have already seen (e.g. **Algorithm 2**) but use long operands split into W sized operands. Using a 16 bit polynomial multiplier on a 16 bit RISC simulator, they demonstrated effectiveness of their software algorithms. They modeled the instruction to work in 4 cycles and gave figures of ECC performance. Also in [39], the same instruction is modeled in an ARM simulator to be single cycle and the effect was a 33 % decrease in ECC execution time. Both the above approaches just modeled the instruction in simulator without considering hardware implementation overheads. Therefore, we wish to extend this approach to do a comprehensive analysis of the same.

In [40] , in addition to MULGF2 other instructions explored were bit-reverse and shuffle. Bit Reverse was introduced to use a multiplier half the word size of the processor. The multiplier first multiplied lower half words of both operands and then the operands were reversed. Again lower half words of operands are multiplied which are actually upper of the original word. Shuffle was used to interleave zeroes for squaring. Also they considered superscalar instruction issue of these custom instructions. Different combinations of bit reverse, shuffle and MULGF2 (they call it $bfmul$) demonstrated speedups for ECC operations. Again here no consideration has been given to area, latency overheads and power requirements.

Chapter 3 Analysis and design for ARM ISE

As we have already stressed, the most important aspect of Instruction set extension is an effective hardware software co-design. In this chapter we present our methodology and arguments for design of a hardware accelerator that could serve as an ARM instruction set extension (ISE).

We consider multiple applications which in our case are ECC and FFT and also algorithms to implement them. We also consider costs and practical aspects of instruction set extension. This is in contrast to the usual approach of predicting performance gains by just modeling the instructions on a processor simulator for instance ([38] [39] [40]). We measure performance by implementing the hardware rather than simulating.

Our way is more similar to FPGA prototyping of a hardware module but again we are measuring performance over an existing industrial platform. This is a marked difference from the usual way to implement a FPGA prototype of a fast hardware algorithm without considering the actual overheads involved in interfacing the module to a processor. The fallacy of this approach was brought out by [15], where the authors demonstrate how the tendency to neglect integration aspects of high speed coprocessors results in exaggerated speed-ups. The performance of these coprocessors are limited by the data bus transfer width and the on-chip data travel time. Thus we would take a holistic view of the system along with algorithms that would run on it, features of ARM instruction set and the trade-offs involved.

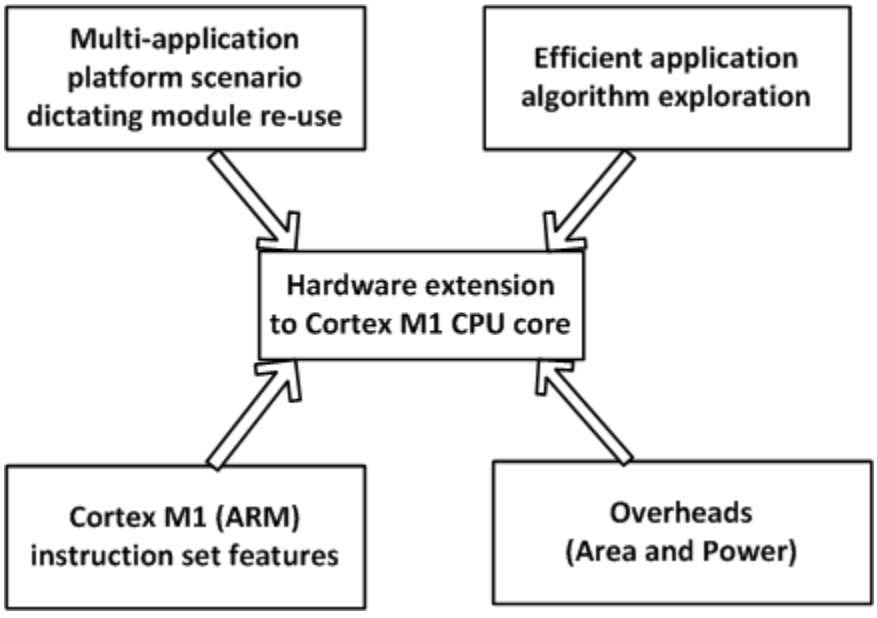


Figure 6 Our approach towards instruction set extension on ARM Cortex M1 soft-core.

3.1 Analyzing the application algorithms

Analysis of the algorithms used for Elliptic curve cryptography and FFT makes us aware of the computation-intensive blocks in it. As already shown by **Table 1**, an ECC implementation is made of number of binary field operations. These operations are namely addition, squaring, multiplication and inversion. Multiplication and squaring have to be reduced by a characteristic polynomial for the operation to be complete. Therefore, it makes sense to consider each of these operations individually and define a hardware-software boundary for their execution. It means certain computations that could best utilize existing ARM instruction features are left to be done by software. Similarly, the radix-2 algorithm for FFT implementation is also analyzed.

3.1.1 Binary field addition

Binary addition of elements is a simple logical XOR of elements. ARM instruction set provides EOR (logical XOR) instruction which executes in possibly a single cycle. So addition in our case is addition of two 163 bit operands or two arrays of 6 integer elements each as shown in **Algorithm 1**. This could be done possibly in 6 cycles or less depending on efficient pipelining and the compiler. Therefore, addition was implemented in software.

3.1.2 Binary field multiplication

Between the two algorithms shift-and-add (see **Algorithm 2**) and Right-to-left Comb multiplication (see **Algorithm 3**), the latter provides obvious advantage because of reduced number of operations. The iterations of core loop in step 2 is reduced from m to word-size of processor W (32 in our case) in Comb multiplication. The major part of the algorithm relies on the shift and XOR operation and the barrel shifter of ARM is supposed to do multi-bit shifts in possibly a single cycle. This would mean that **Algorithm 3** with reduced loop count, multi-bit shift operations and XOR should execute fast in ARM software. But there still could be gains achieved by speeding up the core in step 2 of **Algorithm 3**.

3.1.3 Binary field squaring

Squaring could be done easily by generating a look up table to produce interleaving zeroes amongst operand bits as mentioned in **section 2.1.1.1.3**. This procedure was avoided because it would have cost extra memory and unnecessary memory fetches.

The look up table can be created in the following way. Consider a 16-bit number 23. It would be represented in 16 bit binary representation as 0000000010010001 and the

corresponding 32 bit entry which is 'binary square' of this number would be 00000000000000000100000100000001 or 0x00008201 in hexadecimal. For a 163 -bit operand represented by an array of 6 unsigned integers each array member element has to be split into two and corresponding square entry for it should be fetched from the look-up. The **'binary square'** of the entire array which is the operand could be found with 11 lookups (ignoring upper half-word of last element as it is just a padding of zeroes). We would have 2^{16} such combinations which could be a significant overhead. We could also generate a smaller lookup of 2^8 combinations and this would mean 21 lookups for the answer but we would need to split each array element into four. The memory fetches and write may not be a good option in RISC architecture like ARM and also it involves splitting of operands into smaller bit lengths.

3.1.4 Reduction by characteristic polynomial f(z)

As already mentioned, the fast modulo reduction (see **Algorithm 3**) is very optimized with as less as 24 steps for reducing a 324 bit polynomial.The core of this algorithm relies on shift and XOR operation. A very good performance is attainable with ARM instruction set because all it has to do is shifts and later XOR's. Both these instructions possibly being single cycle, with efficient pipelining by the processor, modulo Reduction can execute very fast in software.

3.1.5 Binary field inversion

As we see from **Table 1 (section 2.1.1.3)**, use of Lopez-Dahab projective coordinates reduces number of inversions to just 1. So it was not implemented. We do not expect a single Inversion to be a performance bottle-neck in implementation of **Point Multiplication** and concentrate our efforts on multiplication. As pointed out in [40], more than 93 % of the time is spent in multiplication and squaring. Multiplication takes up more than 87 % of execution time required for Point Multiplication.

3.1.6 Fast Fourier transform (FFT)

The radix-2 decimation in time algorithm was selected for the sake of simplicity in its implementation. Our approach was not acceleration by algorithm optimization but to apply this algorithm in a multi-application scenario. As correctly noted in [22], the bit reverse step in FFT can take upto 10-50 % of FFT execution time. But again, there exist algorithms as in [41] to perform a bit reverse of using XOR's, shift and logical OR in as less as 6 steps. All these operations are single bit in ARM architecture. Although not present in our platform, ARM provides a custom instruction RBIT for single cycle bit reverse of a 32 bit number. It is present in higher platforms like Cortex – M3. So optimization in this area

is no longer a novelty as the higher platform can do N reverses in N cycles or even less. Another area that could be looked upon for speeding up FFT was the high number of complex multiplications. It is N at each stage i.e $(N * \log_2 N)$ in total. Each complex multiplication involves 4 multiplications but as shown in [42] can also be performed in 3.

$$If\ p = a + ib\ and\ q = x + iy \quad then \quad pq = (ax - by) + i(ay + bx)$$

But it can also be done as, $pq = (ax - by) + [(a + b)(x + y) - ax - by]\, i$

Thus performance is highly dependent on underlying multiplier for FFT. We need to consider the performance of trigonometric functions necessary for calculation of twiddle factors.

3.2 Hardware software partitioning

As we see from the above **section 3.1**, the following diagram can show tasks involved in our two applications.

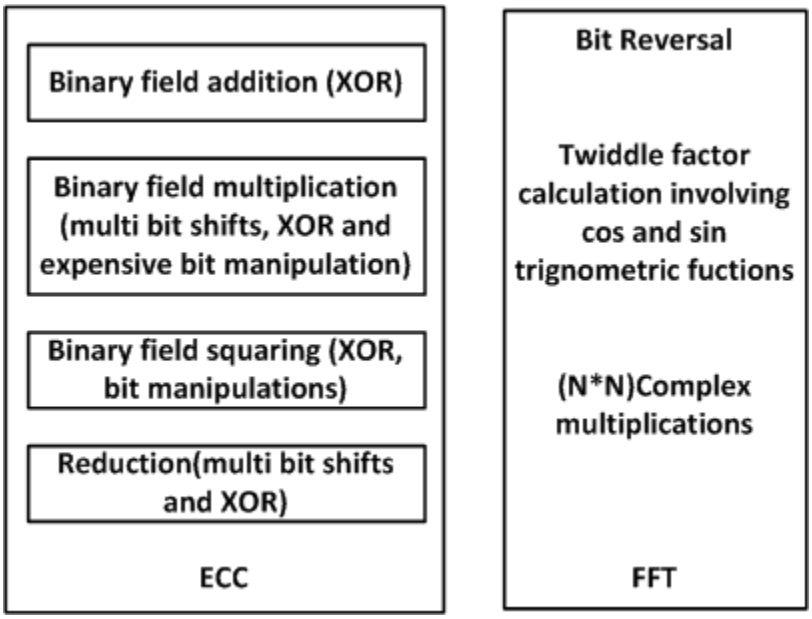


Figure 7 Tasks in ECC and FFT

It should be noted that high computation time for ECC is due to multiple times the internal blocks are used so even moderate savings in any one of them will give considerable savings in ECC point multiplication. Each multiplication and squaring would be followed by reduction to make multiplication complete. Also, complex multiplications in case of FFT are floating point operations which take more cycles.

Thus the following tasks would have maximum support from ARM architecture and would perform better left in software

1. Addition (only EOR)

2. Reduction (logical and arithmetic shifts and EOR)

3. Bit Reversal (assuming availability of RBIT)

4. Floating point operations (assuming floating point support which is a norm)

Some degree of hardware support could be attempted to the following functions with general purpose utility in vision -

1. Binary field multiplication and squaring (expensive bit manipulations)

2. Trigonometric functions for FFT

3. Complex multiplication in FFT

Since we chose Projective coordinate representation, it is important to speed up binary field multiplication as it heavily depends on it (number of multiplications lower if affine coordinates used but number of intensive inversions more).

A small snippet of compiler generated assembly code (THUMB 2) of one of the reduction algorithm steps supports our partition criteria. As seen highlighted, most of assembly instructions generated are shifts and XOR's. The rest are memory read/write and common ADD instructions.

```
product[i-5]=product[i-5]^(l<<4)^(l<<3)^l^(l>>3);//C statement
 1dc: 6dbb        ldr   r3, [r7, #88]     ; 0x58
 1de: 1f5a        subs  r2, r3, #5
 1e0: 6dbb        ldr   r3, [r7, #88]     ; 0x58
 1e2: 1f59        subs  r1, r3, #5
 1e4: 1c3b        adds  r3, r7, #0
 1e6: 3310        adds  r3, #16
 1e8:0089      lsls  r1, r1, #2
 1ea: 58c9        ldr   r1, [r1, r3]
 1ec: 6e7b        ldr   r3, [r7, #100]    ; 0x64
 1ee:011b      lsls  r3, r3, #4
 1f0:4059      eors  r1, r3
 1f2: 6e7b        ldr   r3, [r7, #100]    ; 0x64
 1f4:00db      lsls r3, r3, #3
 1f6:4059      eors  r1, r3
 1f8: 6e7b        ldr   r3, [r7, #100]    ; 0x64
 1fa:4059      eors  r1, r3
 1fc: 6e7b        ldr   r3, [r7, #100]    ; 0x64
 1fe:08db      lsrs  r3, r3, #3
 200:4059      eors  r1, r3
 202: 1c3b        adds  r3, r7, #0
 204: 3310        adds  r3, #16
 206:0092      lsls  r2, r2, #2
```

3.3 Why MULGF2 instruction set extension

As seen in the last chapter, a scalable dual field multiplier [37] can be used for multiplication over both finite and binary fields. This multiplier is configurable using a select signal to make it a polynomial or normal integer multiplier. This is because polynomial multiplication is basically multiplication with carry-less addition of partial products. The design of the hardware multiplier is further elaborated in **section 3.5**. The authors in [38] [37] demonstrated this multiplier for speeding up binary field multiplication. This was a candidate for hardware implementation because a polynomial multiplier can accelerate ECC while it can be used for a general purpose integer multiplication. If we implement a 32-bit version of this multiplier, the operation would take 32 cycles. A simple low area integer multiplier (present in Cortex M1) also works in the same amount of cycles. Therefore, by implementation of this dual multiplier we would be able to ascertain costs of modifying integer multiplier in Cortex M1 micro-architecture to make it a dual field multiplier.

The instruction to perform multiplication over binary fields is popularly called as MULGF2 [38] [37]. Thus a MULGF2 instruction would take two 32 bit numbers and give a 64 bit polynomial product also called as Galois field product. As detailed ahead in the chapter the same MULGF2 can also be used to perform squaring in a faster way. We thus settled to evaluate the effects of MULGF2 extension to ARM instruction set in our platform Cortex M1. As argued we aim to ascertain costs associated with it and not just predict performance figures with simulation. The hardware extension to carry out this instruction can be configurable to perform integer multiplication also. Thus it broadly satisfies our approach of multi-application design.

The following ISE's were not chosen for reasons outlined with them:

Reverse a bit – can be commonly used for both applications but already supported by ARM

Shuffle – Not general purpose

Trigonometric functions – again applicable to only FFT here

3.3.1 Algorithms for Binary field operations using MULGF2

Word level algorithms to perform binary multiplication and squaring exist as seen in [38]. They are described below. The algorithms basically split the m bit operand into blocks of word size W. Then, multiplication is done word by word and finally combined to give a $(2m-2)$ bit result. Also, the binary operand representation is changed to maintain similarity to the authors in [38] i.e equation (1) can be changed to

$$a[z] = \{ a[t-1], a[t-1] \ldots \ldots . a[1], a[0] \} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (10)$$

First element of the array represents MSB bits of 163 –bit operand and upper 29 bits are padded with zeroes similar to our earlier representation. This does not affect any other aspects of operation.

Here, $\otimes$ is used to denote word level MULGF instruction while $\oplus$ is used to denote exor operation. The tuple (u, v) represents a long word of size $2W$ and degree $(2W-1)$. The actual value is $(u(t)\ t^W + v(t))$ which is u and v as a single $2W$ bit value.

Algorithm 6 Pencil and paper method (**see Algorithm 1** [38] **, page 462)**

Input: Binary polynomials $p(z)$ and $q(z)$ of maximum degree $m-1$

Output : $r(z) = p(z) \cdot q(z)\ mod\ f(z)$

1. $r(z) = 0$
2. **for** i from 0 to $t-1$ **do**
   $u \leftarrow 0$
   for j from 0 to $t-1$ **do**
   $(u, v) \leftarrow r[i+j] \oplus (p[j] \otimes q[i]) \oplus u$
   $r[i+j] \leftarrow v$
   end for
   $r[t+i] \leftarrow u$
   end for

Algorithm 7 Comba's method for binary polynomials **(see Algorithm 2** [38] **, page 462)**

Input: Binary polynomials $p(z)$ and $q(z)$ of maximum degree $m-1$

Output : $r(z) = p(z) \cdot q(z) \; mod \; f(z)$

1. $(u, v) \leftarrow 0$

2. **for** i from 0 to $t-1$ **do**

    for j from 0 to i **do**

        $(u, v) \leftarrow (u, v) \oplus (p[j] \otimes q[i-j])$

    end for

    $r[i] \leftarrow v$

    $v \leftarrow u, u \leftarrow 0$

  end for

3. **for** i from t to $2t-2$ **do**

    for j from $(i-t+1)$ to $t-1$ **do**

        $(u, v) \leftarrow (u, v) \oplus (p[j] \otimes q[i-j])$

    end for

    $r[i] \leftarrow v$

    v ← u , u ← 0

  end for

4. $r[2t-1] \leftarrow v$

Algorithm 8 Word-level squaring **(see Algorithm 3** [38] **, page 462)**

Input: Binary polynomial $p(z)$ maximum degree $m-1$

Output: Square $r(z) = p(z) \cdot p(z) \; mod \; f(z)$

1.**for** i from 0 to $t-1$ **do**

    $(u, v) \leftarrow (p[i] \otimes p[i])$

    $r[2i] \leftarrow v$

    $r[2i+1] \leftarrow u$

  end for

3.4 Expected performance with MULGF2 hardware extension

This section describes the expected gains in the following operations by use of MULGF2 instruction.

3.4.1 Binary finite field multiplication

It can be seen that both **Algorithm 6** and **7** have ($t * t = t^2$) MULGF2 operations. So with an effective MULGF2 design, the number of cycles required for point multiplication comes close to ($t^2 * Number\ of\ cycles\ for\ one\ MULGF2$) . In contrast to step 2 of **Algorithm 3**, the expensive bit manipulation operation at the end of the loop is avoided altogether.

The interesting aspect is to check the effect when the hardware multiplier is assumed to be in processor data path. This would be a true instruction extension and give true theoretical speed up. Step 2 of **Algorithm 3** in software can be shown by following piece of code.

```
    for bit_postion from 31 down to 0
1.      for i from 0 to t-1
            If ( bit_position of A[i] contains 1)
                for j from 0 to t
                    product[ i + j] ← product[ i + j] ⊕ sb[j]
                end for
        end for

2.      for i from t down to 0
            sb[i]=sb[i]<<1 ⊕ sb[i-1]>>31;
        end for

        sb[0]=sb[0]<<1;
    end for
```

There are $32 * t * t$ XOR's in 1st step and $32 * t$ XOR's in step 2 along with shifts for bit manipulations.

This was replaced by **Algorithm 6** and **Algorithm 7,** which contain ($t * t$) MULGF2 calls and execute in $t * t * 32\ cycles$. There are ($2 * t * t$) XOR operations and no bit or shift operations.

If $t = 6$, as in our case the number of XOR's in **Algorithm 6** and **Algorithm 7** is 1244 against 1344 from the above core. With no bit manipulations, efficient compiling and pipelining by the processor (MULGF2 if assumed to be in instruction set) both algorithms should execute faster than the core used above. Note, we have not considered load and store operations for the above calculations because different addressing modes may be possible when MULGF is included in instruction set. But the number of load and stores in the above core is much more than in **Algorithm 6** and **Algorithm 7**. If such gains can be achieved over a single binary field multiplication operation, ECC would be considerably accelerated.

3.4.2 Binary finite field Squaring

As seen from algorithm 8 squaring could be finished in just $(t * Number\ of\ cycles\ for\ one\ MULGF2)$ i.e $(t * 32)$ instead of the number of cycles taken by the above software core. For benchmark purposes due to reasons mentioned already (see **section 3.1.3**) we use the same multiplication algorithm to calculate squaring performance in software.

3.4.3 FFT

The performance of FFT would remain more or less the same because our dual multiplier behaves same as simple integer multiplier. As pointed out in the next section, modifying the same integer Multiplier micro-architecture to a Dual multiplier for MULGF2 extension adds negligible area and delay to existing one. So we now focus on ECC and underlying binary field arithmetic because that would give us measurable results while FFT would just behave the same.

3.5 Hardware design of multiplier

This section explains the design of Shift-and-add integer multiplier and how it is modified to make it a configurable dual field multiplier or a multiplier which can perform both integer and polynomial multiplication

3.5.1 Shift-and-add integer multiplier

The simplest kind of multiplier design used for multiplication in binary format is the Shift-and-add multiplier. Consider B as the multiplier operand and A as the multiplicand. Then we get the Result $(A * B)$ by adding B number of shifted partial products of A (A or 0). This is very similar to school book multiplication.

For e.g. two unsigned 4 bit numbers 9 (1001) and 7 (0111) would be multiplied as

1001 → A

0111 → B

1001

1001

1001

0000

01100011 (63)

Thus for hardware implementation, there are n steps for multiplication of two n-bit numbers. The multiplicand A is a 2n bit register as it is left shifted by 1 at each step to form a partial product. The lower n bits of register A are initialized to original operand B and upper bits to 0. Register B is of width n bits and initialized to original multiplier operand B. The final result register Product (2n bits) is initialized to 0. At each step, register B (n bits) is right-shifted by 1 and its LSB is checked. If the LSB of B is 1, value in register A is added to accumulator Product. Thus at the end of n cycles we have the result of multiplication in the 2n bit register Product.

The above design could be optimized with respect to area. The first observation is that on each shift to the multiplicand register A zeroes are inserted at LSB. So LSB digits of accumulator Product once formed do not change as only zeroes get added to them. Instead of shifting multiplicand left, we can shift product right at each step. Thus A remains fixed relative to the Product. We would need only a n-bit adder as opposed to a $2n$ bit adder in earlier case. Only left half of the register Product would be changed. Also, the Product register in earlier case has free space of n bits at start which starts filling up while multiplier register B has n bits at start and gets emptied. Thus in the final hardware design, the register Product and multiplier operand register B are combined. Product is just n-bit wide and the final result is made up of Product and B.The final operation can be shown as follows:

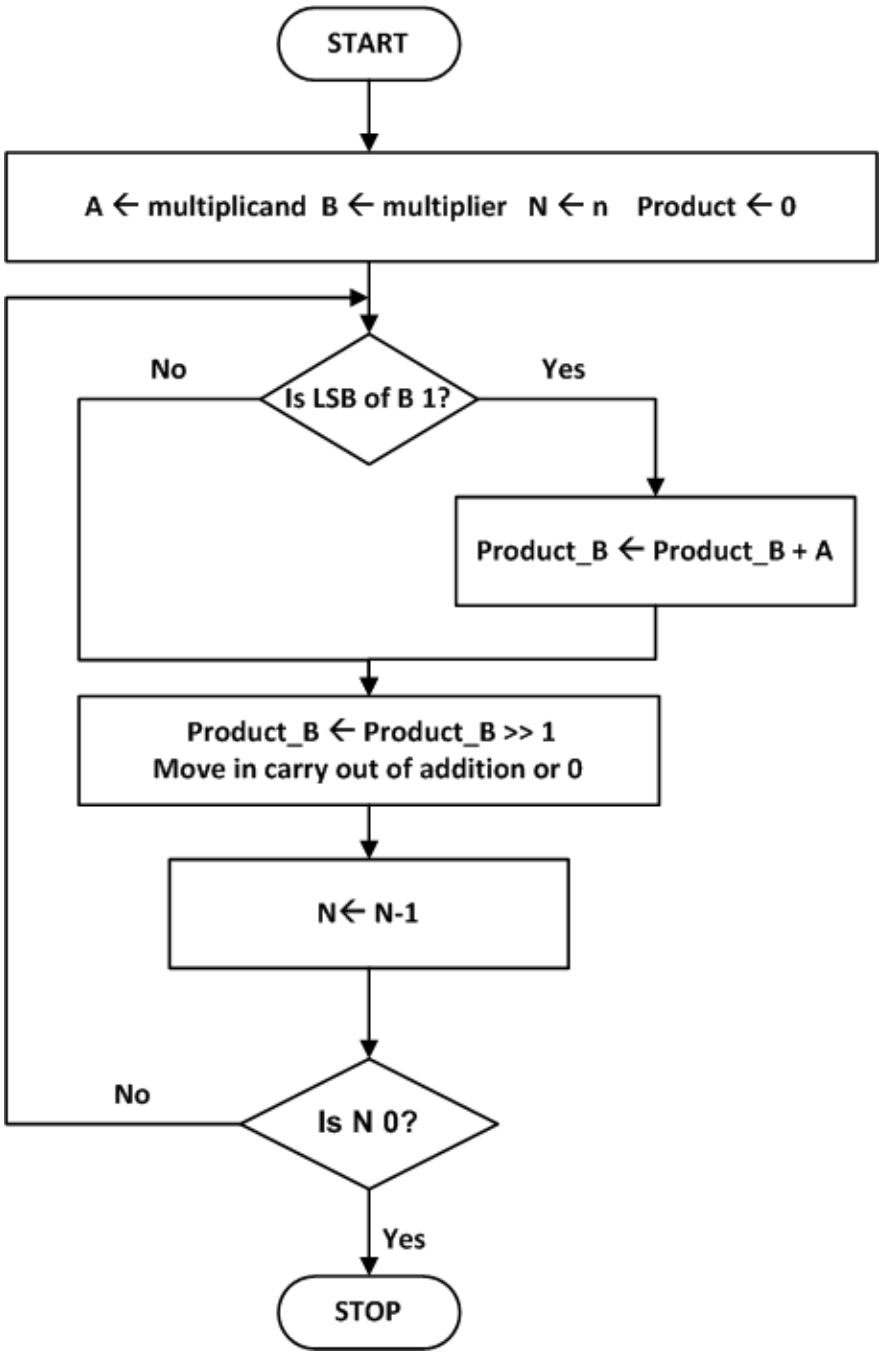


Figure 8 Shift and add multiplication algorithm

3.5.2 Shift-and-add dual field multiplier

We can modify the above design to make it a Shift-and-add dual field multiplier. We see from the above design n in our case is 32. The 32-bit adder required for the above multiplier is made up of 32 dual field full adders. Each dual field adder (DFA) [37] is configurable using $fsel$ signal to give addition with or without carry. Thus we can perform both polynomial and Integer multiplication using this designed multiplier. The Dual field adder is shown in the following diagram (see figure 1 [38] page 461).

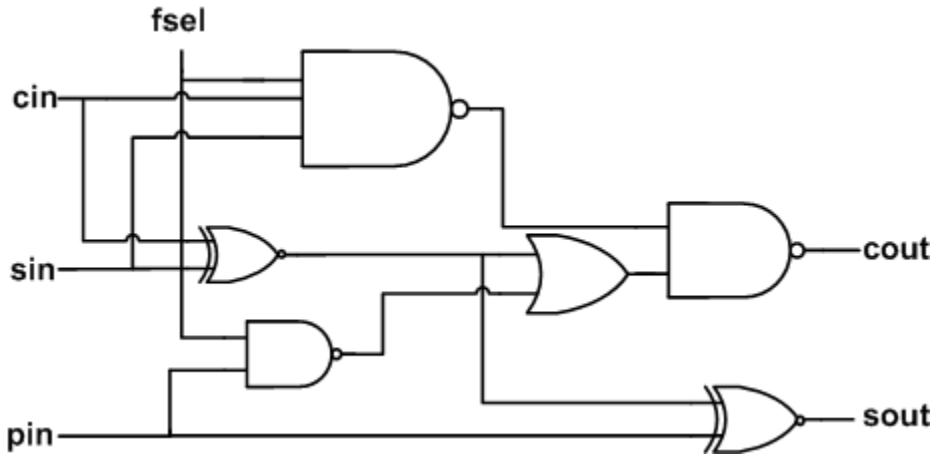


Figure 9 Dual field adder

Thus when $fsel = 1$, the multiplier gives result of integer multiplication of A and B. But when $fsel = 0$, each full adder asserts carry out (cout) as zero in all cases and therefore the addition of partial products is without carry. This is basically polynomial multiplication which is required for multiplication over $GF(2^m)$.

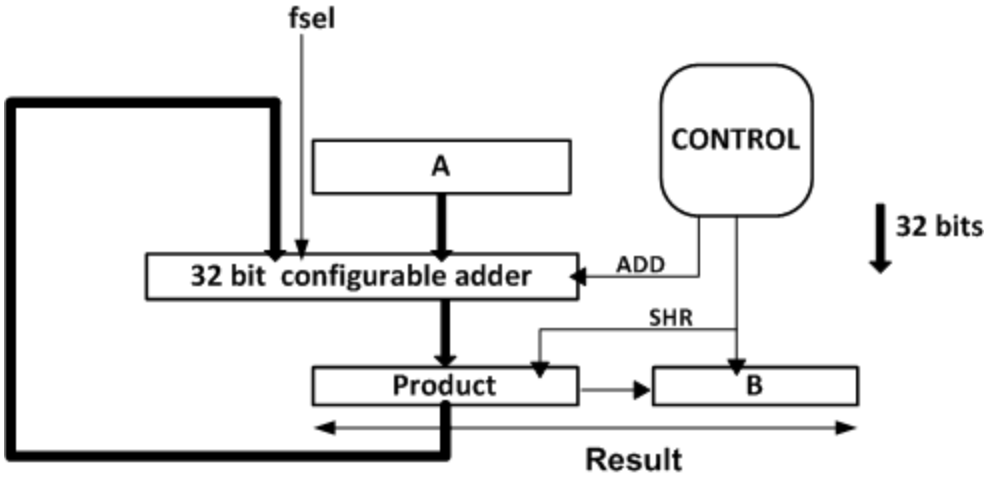


Figure 10 32-bit Dual field multiplier

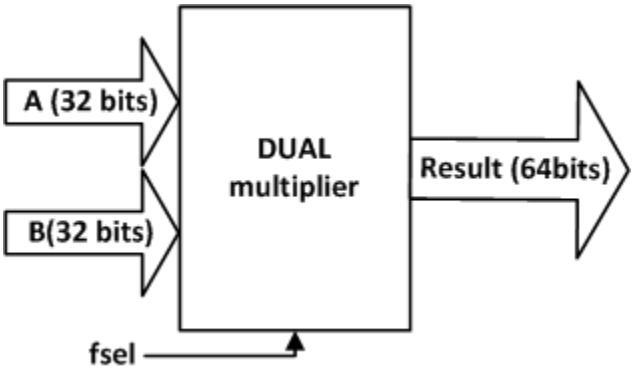


Figure 11 High level block diagram of Dual field multiplier

The hardware dual field multiplier performs both integer and polynomial multiplication depending on the fsel signal we give. CONTROL basically denotes that , a SHR signal is given to combination of registers Product and B on each clock cycle, while ADD signal is given only if LSB of register B is1.

Chapter 4 Implementation

This is the era of Programmable system chip (PSC) in which designers and manufacturers desire rapid development, configurability all on a single chip. Cryptography demands flexibility of the software and reliability of the hardware which is achieved by the help of a soft core general purpose processor in FPGA based design.

We choose a platform provided by ACTEL called as FUSION for our purpose. It makes sense to evaluate ECC behaviour over ARM core based on ARM7TDMI, which is arguably the most widely used 32 bit microcontroller platform. FUSION uses Cortex M1 soft-core, the first ARM core specifically designed for FPGA implementation. On chip flash is provided to provide storage for the microcontroller core. Also ability to add peripherals to a standard bus interface like AMBA helps in single-chip design. Some of the main advantages are attributed to the fact that its a FLASH based FPGA. This accounts for reduced power consumption and ability to retain configuration memory. These FPGA's retain configuration data in on-chip flash unlike other FPGA's which need external flash. They are 'live-at-power-up' i.e. no external data needs to be supplied on power-up. Also if considered for cryptographic applications these power-up configuration data may be subject to hacks and thus our platform provides safe storage [5].

4.1 Cortex M1 – The ARM soft-core

Cortex M1 is a 32-bit ARM7 family core which is available without any royalties or fees to be implemented on an FPGA. It is a simple 3-stage pipeline,Von-Neumann processor. According to Cortex M1 handbook [43], an advanced configuration with provision for 16 interrupts and enhanced debug occupies about 11000 tiles out of 30000 available. This leaves us enough space to add a AMBA bus system, memories and other accelerators. AHB reads happen without WAIT states while writes go through a one entry write buffer to avoid WAIT states. WRITE transaction is immediately completed on the bus unless when write buffer is used. if bus waits for data during buffered store [43]. It also features a tightly coupled memory (TCM) interface which allows low latency instruction and data access with the help of Instruction TCM (ITCM) and data TCM (DTCM). The processor implements a subset of Thumb-2 (ARMv7) architecture called ARMv6-M [43].

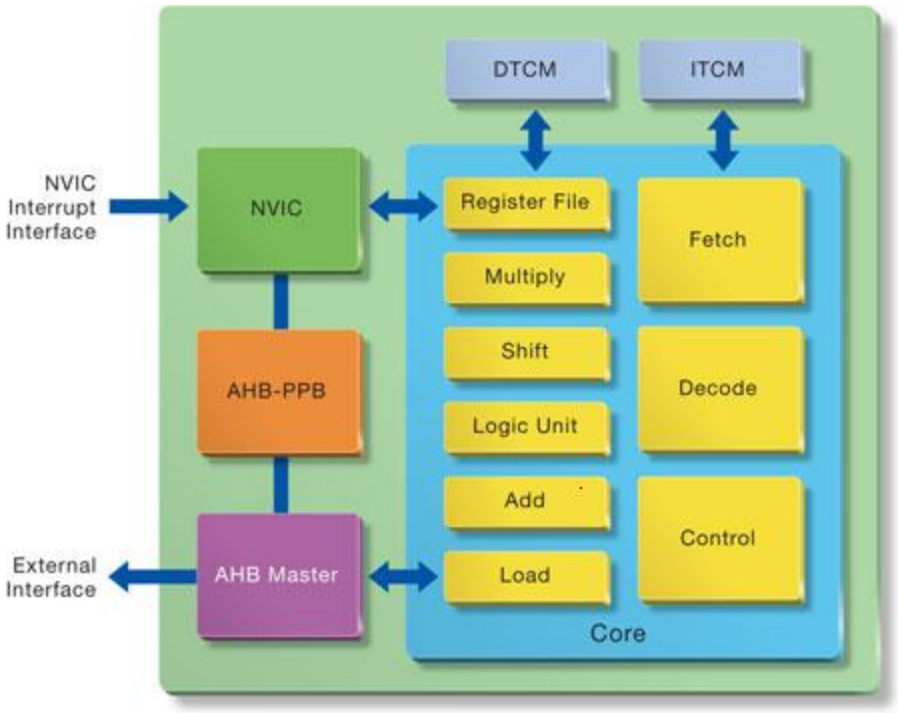


Figure 12 Cortex M1 block diagram

(Image from **cts/mpu/Corhttp://www.actel.com/produtexM1/** last accessed on 28/09/2011)

4.2 AMBA

AMBA (Advanced Microcontroller Bus Architecture) is an open bus standard used for SoC systems developed by ARM. AMBA helps in creating a framework for SoC development enabling IP reuse. Different teams and vendors can work on peripheral IP's and combine to form a complete SoC solution. All that is needed is a bus wrapper above the custom IP. The AMBA 2.0 bus system contains the following components -

Advanced High performance Bus (AHB) [23] : It's a single clock cycle data transfer bus which is used to connect high speed and high performance modules. It supports burst mode data transfers and split transactions. A slave with large response latency can free the bus for other transfers. It supports multiple masters and also provisions for 64 bit or 128 bit operations. AHB transfers have no wait states between master and slave because of pipelined structure. Address/control signals and data transfer for two different transfers can happen in the same clock cycle. For e.g. in the above system we connect the ARM CPU (processor core) as the master and memory controller as AHB slaves. We use AHB-lite a version of AMBA for our project which is a simplified version with one bus master.

Advanced System Bus (ASB) [23]: It's an alternative to AHB for medium performance systems and also supports burst mode data transfers. We do not make use of ASB in our hardware.

Advanced Peripheral Bus (APB) [23]: It's a low power bus used to connect comparably slow peripherals. It has less interface complexity and optimized for minimum space power. When AHB and APB are both used in a system we use a AHB to APB bridge which is mapped as an AHB slave. Components such as UART's, Timer's custom blocks like divider are connected to the APB.

More information about the AMBA 2.0 specification can be found in [23].

4.3 Hardware and software development environment

4.3.1 ACTEL Libero IDE

ACTEL provides an **integrated IDE 'Libero'** [44] for hardware development on its FPGA's which includes FUSION platform which we use. It enables us to choose standard IP blocks in RTL format from a proprietary repository. The ARM Cortex M1 core also appears as an IP in this environment. This reduces development efforts to a considerable extent as IP blocks of AMBA bus structure and standard peripherals like UART's, Timers, memory controllers etc are available. The GUI based tool allows us to drag these IP blocks into a canvas (SmartDesign). The on-chip connection between these hardware blocks can be done automatically or even manually on this canvas. This simplifies the glue logic development between RTL modules on a complex processor based SoC system. Also, it allows defining memory map for the processor and fixing base addresses of peripherals which are later used to write peripheral device drivers. The peripherals offered by the IP repository are verified for inter-timing accuracy and are AMBA compliant.

The custom hardware block we design can be easily attached to other components on the canvas by writing a bus compliant wrapper around it. The entire design flow from hardware design, verification and simulation till synthesis can be carried out in Libero. In addition to this, floor planning for FPGA, constraints definition, timing analysis , 'place-and-route' and programming file generation can be done in the integrated 'ACTEL designer'. SmartPower [45] used in ACTEL 'designer' enables measurement of static and dynamic power consumption of a design and estimates power consumption of individual components like cores, IO's, memory etc. Verification of the custom hardware peripheral developed for a ARM processor based SoC can be done by 'Bus functional model' (BFM) early in the design cycle. BFM is a cycle accurate model of the processor which can be used to perform memory read/writes over an AMBA based system at any stage of development.

BFM scripts allow us to perform operations like poll at a memory location, insert wait cycles, compare read memory values with constants and so on. Thus the correctness of operation of a peripheral along with its timing compatibility with the processor can be verified by BFM [43].

4.3.2 SoftConsole

SoftConsole [46] is used to produce application software development to be run on the ARM softcore. SoftConsole includes a internal GNU CodeSourcery ARM compiler. It also has a debugger that allows access to processor registers and memory contents. Also, the application can be programmed into FLASH which can be used as instruction memory of the processor.

4.4 Overview of Implemented system (hardware)

4.4.1 Complete system on FPGA

Cortex M1 is provided as a black box encrypted RTL for use in our FUSION FPGA platform. Adding the above 32 bit hardware dual multiplier in the processor data path is not possible although that would have given the best results. So the multiplier design was done and added as an external peripheral to the Cortex M1 soft-core over AMBA protocol. This allowed us to test the peripheral in a loosely coupled configuration

The entire system was designed with Cortex M1, memory blocks, GPIO and UART peripheral by us as shown in the following figure. The analog block IP provided ACTEL was added for future use and not used in this thesis. The blocks shaded are our IP's and the code is attached in the APPENDIX. The address space of Cortex M1 is 4 GB, the address bus being 32 bit. 16 AHB slaves have to be addressed utilizing 4 bits and 28 bits are used to address AHB peripherals internally. One of the AHB slaves AHB2APB bridge, is further used to interface APB bus. The AHB2APB bridge is the master of the APB bus and controls read/writes to all APB devices. So 28 bits are used to address 16 APB peripherals. 4 bits out of those are used to address 16 possible APB slaves generating a PSEL signal for each APB device. The rest 24 bits are used as PADDR (23 : 0) to address within an APB peripheral as shown. The base addresses of all AHB and APB peripherals are allocated in the Libero IDE. The internal registers of peripherals can be accessed using offsets to the base address of the peripheral. Since the multiplier takes about 32 cycles it was implemented over APB rather than AHB which is high speed bus. The block shown in dotted line is another possibility to add our custom IP to high speed AHB bus.

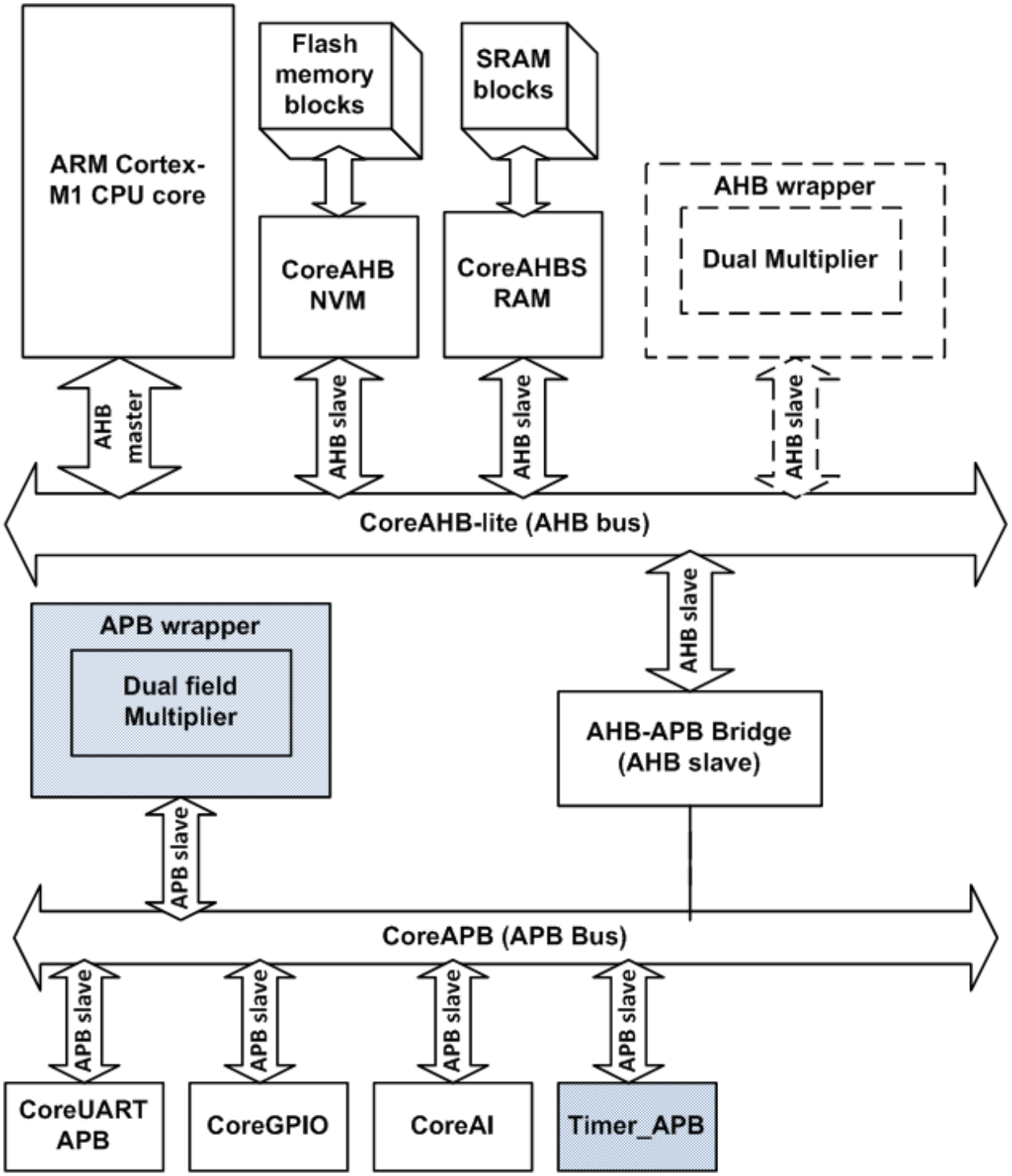


Figure 13 System implemented on FPGA

4.4.2 Dual field multiplier with APB wrapper

An appropriate wrapper has to be written around the dual field multiplier described in **section 3.5.2** to interface it with APB. So the final hardware design with the APB wrapper is shown in the following block diagram and called as Multiplier_APB.

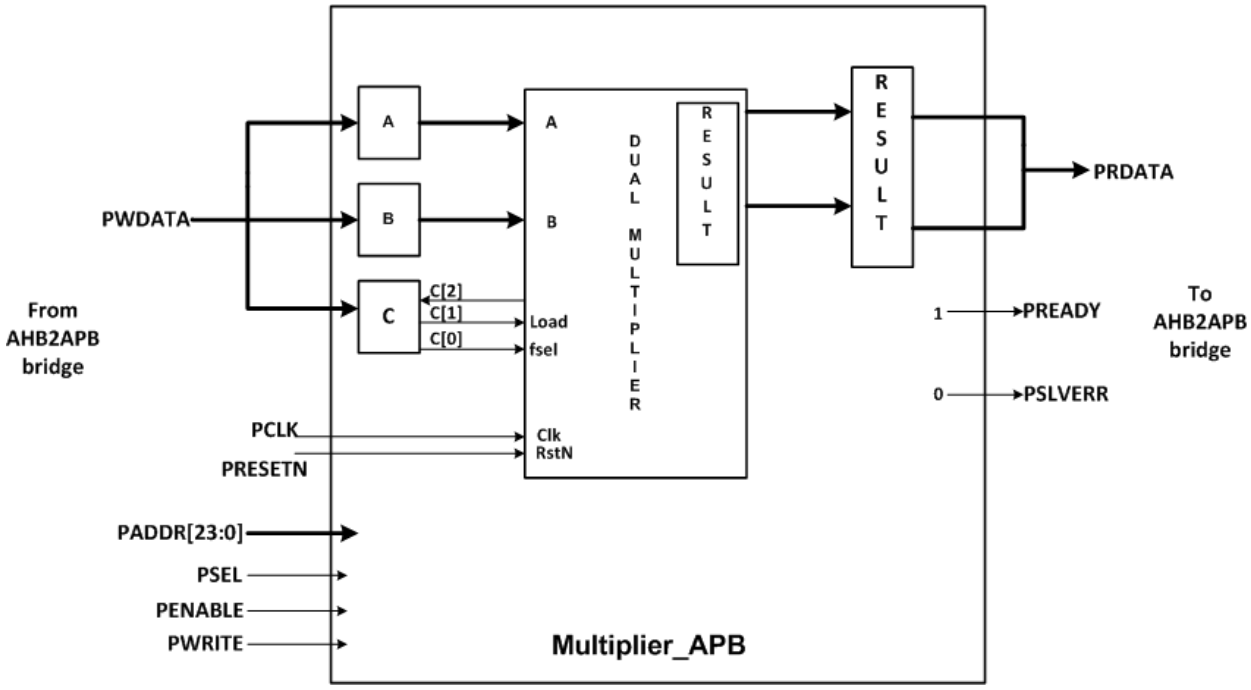


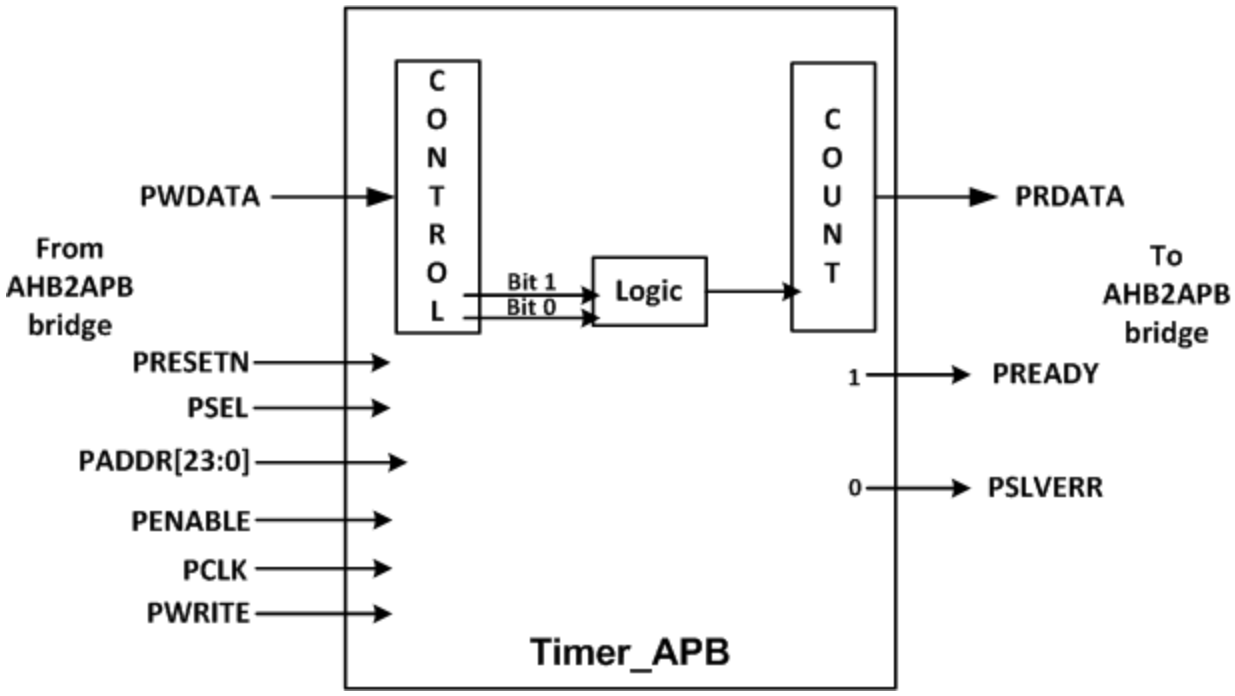
Figure 14 Dual field multiplier with APB wrapper (Block diagram)

The figure shows major signals and registers. Thus read or write to the peripheral is done after selection by **PSEL** and driving **PWRITE** signal (1 for write and 0 for read). 32 bit data Data is placed on **PWDATA** bus which is stable when **PENABLE** goes **HIGH** and remains **HIGH** for one clock cycle. Thus for writing to a peripheral register addressable using **PADDR**, the signals **PSEL**, **PWRITE** and **PENABLE** have to be **HIGH**. Thus, data is written to multiplier peripheral registers **A, B** and **C** by three consecutive **WRITE** operations. The register **C** in our case is used as a Control register with three LSB's acting as control signals. **C[0]** is used as ***fsel*** signal for the multiplier which selects operation of our dual multiplier as integer or carry-less multiplication. **C[1]** is used as a Load or START signal to the multiplier. It indicates that operands for multiplier operation have been loaded into registers A and B and the multiplier operation can start. **C[2]** has been used as a signal to signal end of calculation by the multiplier. Once the RESULT is calculated and placed in a 64 bit register, **C[2]** is made high. After operands are written to multiplier the control register is continuously polled to check **C[2]** is **HIGH** which signals end of operation. When it is found **HIGH**, read operations to 2 addressable locations which form the 64 bit RESULT register is

done. The higher and lower WORD of RESULT are placed on **PRDATA** one after the other. Polling method was preferred over interrupts for the peripheral considering interrupt latency Cortex M1 (more than 30 cycles as said in [28]). In addition to this, some number of cycles would be required for Interrupt service routine instructions. It would not have provided any good gains as our multiplier itself works in 32 cycles. A **PREADY** is used by an APB peripheral to insert WAIT states if required by going LOW. This makes **PENABLE** signal usually **HIGH** for only 1 cycle to extend the **HIGH** state for further number of cycles. In our case, we kept **PREADY** as 1 as we were not using it and writes to FPGA fabric happens with 1 clock cycle operation of **PENABLE**. **PSLVERR** is an optional signal to be implemented and can be used to indicate any failures of read/write. We did not implement it and just made it default LOW because transfers to FPGA fabric from Cortex M1 processor itself on fabric are fool-proof.

4.4.3 TIMER with APB wrapper

Similarly, Timer_APB was designed over APB for performance measurements. It was initially suspected that the hardware Timer IP provided by ACTEL was not accurate and this was designed for further accuracy. It is a simple 32 bit up counting Timer with two addressable registers **CONTROL** and **COUNT**. The bit 0 of **CONTROL** is used to start/stop (000…..1/00…..0) the counter. Bit 1 is used to restart the counter to zero i.e by writing value 2 (0000…..010) to it. As the count starts from 0, the number read from **COUNT** register directly gives number of clock cycles taken by Cortex M1 for a particular task. This is because, it uses the same clock as the Cortex M1 and thus accurate performance measurements of different algorithms were done. The other interface signals have the same functionality as described for the multiplier above.



4.5 Software

The decisions regarding implementation of algorithms have already been elaborated in **section 3.2**. To complete description of the implemented system, some things are repeated here. The algorithms used for binary field arithmetic were implemented in software using Soft Console. It was decided to implement only binary field arithmetic and not the ECC point multiplication as a whole. This is because, we can still calculate performance of a ECC point multiplication using figures gathered from binary field arithmetic and benchmark with existing implementations as shown in the Results section.

For pure software implementation, Right to Left Comb method for Polynomial multiplication **[see Algorithm 3].** Also squaring in pure software was done using the same multiplication algorithm i,e input operands to multiplication algorithm were same. All the above three operations can give us a good estimate of the time required for one ECC point multiplication.

For software-hardware implementation as discussed, **Algorithm 6** and **Algorithm 7** were implemented and evaluated. The word-level polynomial multiplication of the form $(\boldsymbol{a} \otimes \boldsymbol{b})$ which forms core of these algorithms is done using our designed hardware multiplier on the APB bus. That is the function **dual_Multiply** detailed below is called

which in turn acts as the hardware multiplier device driver and passes signals to it. The 64 bit result is read after completion.

The device driver created by us of the hardware multiplier (modified for better readability) is shown below. The operation is the same as in original code. When the function *dual_Multiply* is called it is equivalent of one *MULGF* instruction call.

```
void  dual_Multiply ( unsigned int a, unsigned int b, unsigned int c, unsigned int Result [ ] )
{
    unsigned int i;

    HAL_set_32bit_reg (  MULTIPLIER_APB_BASE_ADDR , MULTIPLIER_REG_OFFSET, a  ) ;
    HAL_set_32bit_reg ( MULTIPLIER_APB_BASE_ADDR  ,  MULTIPLICAND_ REG_OFFSET, b);
    HAL_set_32bit_reg ( MULTIPLIER_APB_BASE_ADDR  , CONTROL_REGISTER_REG_OFFSET, c);
                                    // Control register bit 0 gives fsel and bit 1 gives START signal
    do
    {
     i = HAL_get_32bit_reg ( MULTIPLIER_APB_BASE_ADDR , CONTROL_REGISTER_REG_OFFSET);
    }
    while (!(i&0x4));   // Control register bit 2 polled to check end of multiplier operation

    Result[1] = HAL_get_32bit_reg(MULTIPLIER_APB_BASE_ADDR,RESULT_LOWER_REG_OFFSET);
    Result[0] = HAL_get_32bit_reg(MULTIPLIER_APB_BASE_ADDR, RESULT_HIGHER_REG_OFFSET);
                                                                    //  64 bit Result register read
}
```

Similarly, squaring was done using **Algorithm 8** and using same multiplier.

Binary field addition was done using **Algorithm 1** and Reduction required for multiplication was done using **Algorithm 4** in all cases because of their inherent fast software execution nature.

Thus, in this case the hardware software boundary moves towards hardware and software covers more area. The features of ARM instruction set cause this marked difference compared to **Figure 5.**

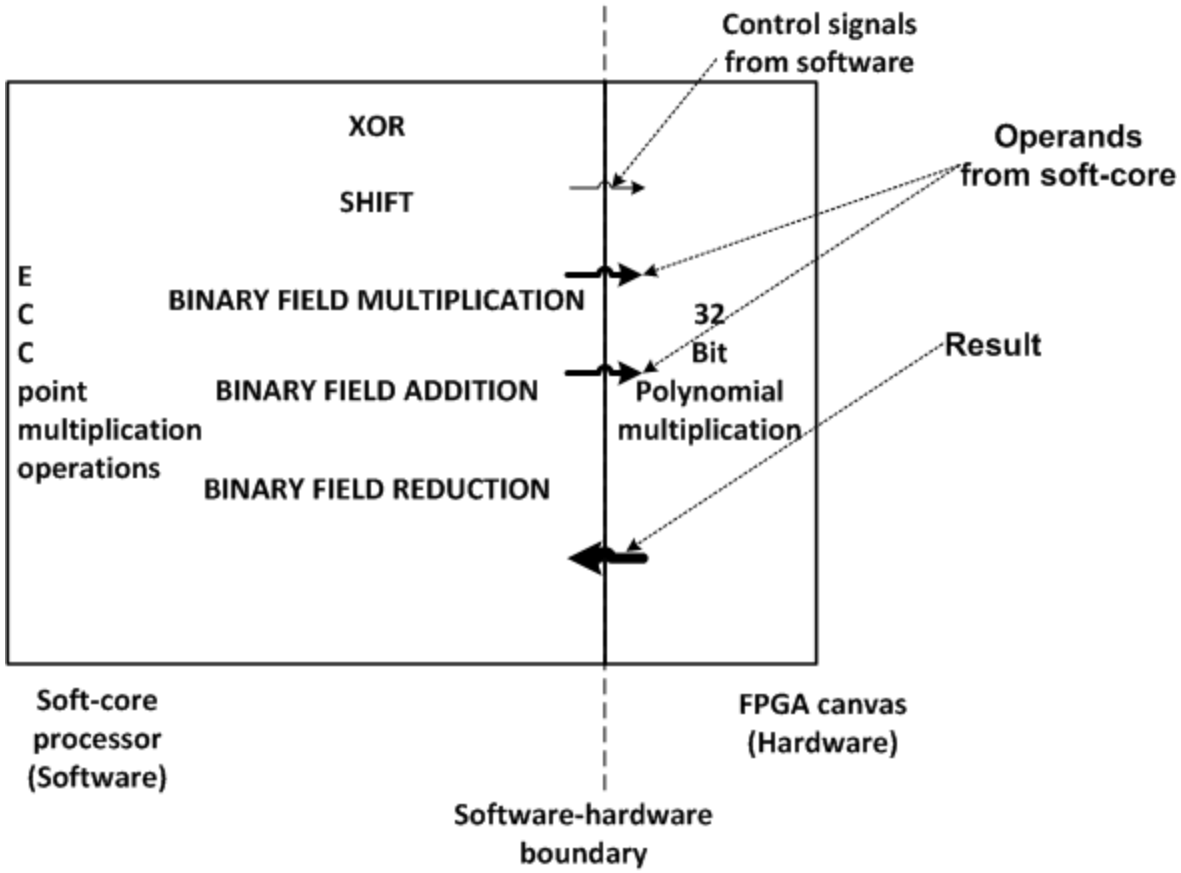


Figure 15 Software-hardware boundary for ECC implementation in our work

Chapter 5 Results and analysis

In this chapter we enlist the observations and results of our experiments. It was a two stage process. In the first stage, the system on FPGA was similar to the one shown in **Figure 13** without the dual field multiplier (designed hardware or APB_multiplier). The software to calculate ECC performance consisted of **Algorithm 1** for field addition, (**Algorithm 3** + **Algorithm 4**) for (multiplication + reduction) and (squaring + reduction). We call this as scheme I for convenience. From the number of cycles obtained for each above procedure, the time required for ECC point multiplication was calculated using **Table 1.** FFT was implemented using radix-2 algorithm.

In the second stage the system on FPGA was as depicted in **Figure 13.** The hardware block 'shift and add' dual field multiplier was added. Now **Algorithm 1** for used field addition, (**Algorithm 6** + **Algorithm 4**) for (multiplication + reduction) and (**Algorithm 8** + **Algorithm 4**) for (squaring + reduction). Note, the operation $(\boldsymbol{a} \otimes \boldsymbol{b})$ **i.e** MULGF2 was carried out using **dual_multiply** function call to hardware peripheral. This way to calculate ECC performance is called as scheme II. Lastly, in scheme III everything was same to scheme II except that (multiplication + reduction) was carried out using Comba's method i.e (**Algorithm 7** + **Algorithm 4**). FFT was not further implemented because the performance of FFT would be same, considering CPU's internal shift and add multiplier. Only ECC provided measurable results, which are highlighted in further sections. The timings for all schemes were measured using Timer_APB custom peripheral.

5.1 Code size

Thus as described the effect of MULGF ISE was tested primarily on multiplication and squaring algorithms. Now, it was interesting to observe the code sizes of all three versions of binary field multiplication. Due to custom ISE, the code size would be ideally expected to reduce. But all three multiplication techniques showed almost same amount of instruction size on being compiled in THUMB2. It could be because of the small size of the multiplication code and compiler optimization enabled. Note Cortex M1 being a Von-Neumann architecture, instructions and data appear in same code memory.

Method or algorithm	Code size
R-L Comb method for field multiplication + Reduction in software (**Algorithm 3** + **Algorithm 4**)	528 bytes
Pencil-and-paper method with MULGF2 for field multiplication + Reduction in software (**Algorithm 6** + **Algorithm 4**)	508 bytes
Comba's method with MULGF2 for field multiplication + Reduction in software (**Algorithm 7** + **Algorithm 4**)	534 bytes

Table 2 Code size comparison with and without ISE

5.2 Clock cycles for Point multiplication $Q = k \cdot P$

Table 1 can be used to calculate number of binary field operations required to calculate ECC scalar point multiplication. Considering $GF(2^{163})$, the highest value of $\lfloor log_2 k \rfloor$ would be 163, simply because k could take all the 163 bits for representation. Note that, the number of computation decreases with degree of k i.e when higher degree coefficients of polynomial are absent. But, we consider degree of the polynomial to be 163 and the numbers of field operations are as follows:

Field additions = $3 * 163 + 7 = 496$
Field multiplications = $6 * 163 + 10 = 988$
Field squaring = $5 * 163 + 3 = 818$
Field Inversion = 1

The average number of cycles for the corresponding operations taken over an average of 50 executions is listed in the table below.

Method or algorithm	Number of cycles
R-L Comb method for field multiplication + Reduction in software (**Algorithm 3** + **Algorithm 4**)	121
Pencil-and-paper method with MULGF2 for field multiplication + Reduction in software (**Algorithm 6** + **Algorithm 4**)	9055
Comba's method with MULGF2 for field multiplication + Reduction in software (**Algorithm 7** + **Algorithm 4**)	10167
Field Addition in software (**Algorithm 1**)	6
Field squaring with MULGF2 + Reduction in software (**Algorithm 8** + **Algorithm 4**)	1543

Table 3 Cycle count for binary field operations using all algorithms

Thus the approximate number of cycles (in thousands) required for ECC scalar multiplication $\boldsymbol{Q = k \cdot P}$ by different techniques is listed below (using **Table 1** and **Table 3**). As already mentioned field inversion has not been considered. Field addition and reduction have been always done in software in all the schemes listed below.

Operation / **Scheme used**	**Multiplication**	**Squaring**	**Addition**	$\boldsymbol{Q = k \cdot P}$ (approx)
I	119.548	98.978	2.976	221.504
II	8946.34	1262.174	2.976	10211.49
III	10044.996	1262.174	2.976	11310.15

Table 4 Cycle count of ECC point multiplication with different schemes

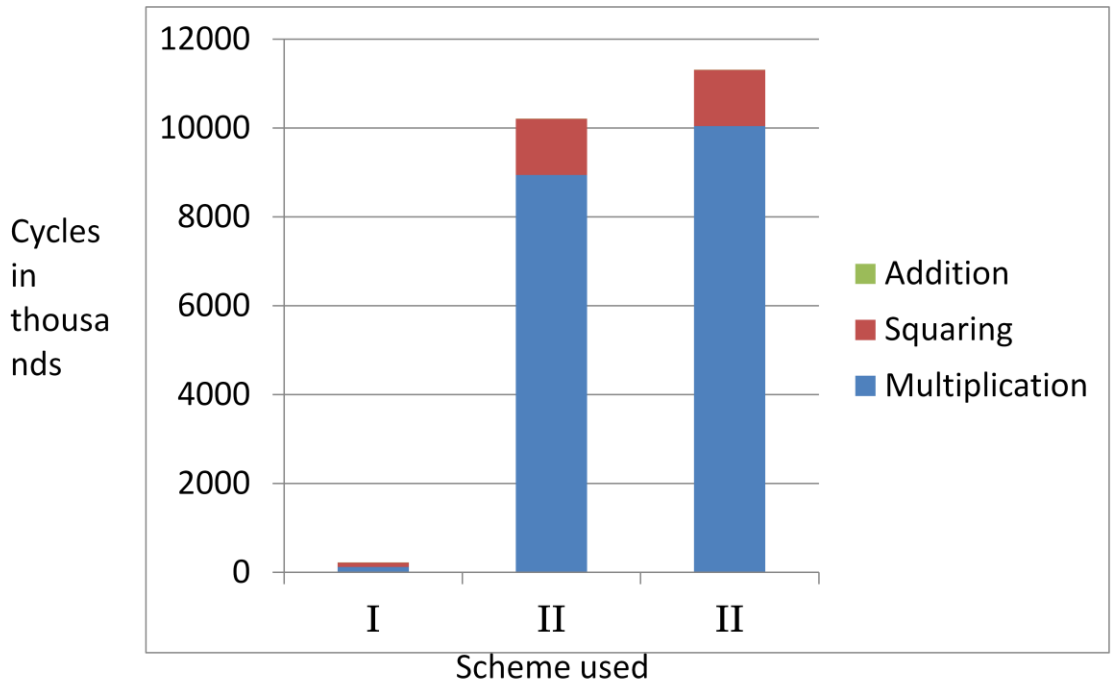


Figure 16 Graphical representation of ECC point multiplication cycle count for all schemes

The current working frequency of the system is 20 MHz while the design was analysed using SmartTime [47] to work correctly upto a frequency of 50 MHz. Then the critical path of **Cortex M1 – AHB – AHB2APB bridge – APB_Multiplier** comes into picture. The slack time for various paths at 20 MHz and 50 MHz can be shown as follows. They are all positive indicating they satisfy timing constraints.

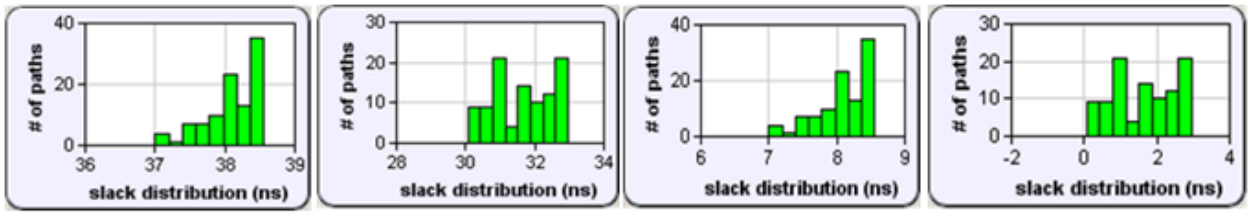


Figure 17 Slack for Cortex M1 to dual field multiplier path at 20MHz and 50MHz

I and II show number of paths and their slacks between 'AHB2APB bridge to APB-multiplier' and 'Cortex M1 to AHB2APB bridge' at 20 MHz respectively. III and IV show the same at 50 MHz respectively.

Using number of cycles from **Table 1** , the following table gives approximate execution time of ECC scalar point multiplication operation for the three above schemes.

Scheme	At 20 Mhz	At 50 MHz
I	0.5 secs	0.2 secs
II	0.56 secs	0.225 secs
III	11-12 msecs	4.43 - 5 msecs

Table 5 ECC point multiplication time for different schemes

The figures are surprising and not expected as we can see the, hardware schemes are taking more than 10000k cycles to perform a single scalar point operation $\boldsymbol{Q} = \boldsymbol{k} \cdot \boldsymbol{P}$. This is because the binary field multiplication **Algorithm 6** and **Algorithm 7** are taking $9000 - 10000$ cycles to operate. Similarly time required by **Algorithm 8** for squaring is very high. On further analysis, and breaking down of the code it was determined that one function call to **'dual multiply'** or MULGF instruction is taking around 300 cycles. This is very high overhead because the hardware operation just takes 32 cycles after operands are written to it. The device driver is taking up 10 times as much time as the device itself. On further profiling, it was determined that two consecutive peripheral register access have about 20-30 cycles latency in between. That is, two consecutive peripheral read or writes have 20-30 **WAIT** states.

On query to ACTEL support, they replied saying the AHB2APB bridge inserts **WAIT** states between two peripheral register accesses. This is to compensate for software access to FPGA fabric. As AHB2APB was an encrypted RTL, similar to other ACTEL provided cores, it was not possible to predict this behavior at earlier stage. Even though we do not insert any WAIT states from our peripheral, the AHB2APB is an AHB peripheral and can insert WAIT states. By AMBA specification, an APB peripheral register access without WAIT states should finish in 2 cycles. In this case, it is taking 10-15 times more. The support team explained it as an on chip communication latency of Cortex M1, which can be in the order of microseconds to even milliseconds. They had no further documentation

explaining these FPGA fabric communication latencies. The time between the software load/store instruction in Cortex M1 and actual load/store from fabric appears to be around 10 cycles. In our case one peripheral access was taking 1.5 microseconds at 20 MHz. This is device-specific and can even vary between two boards of same type. Thus clearly demonstrated in this case, on-chip communication overheads can nullify any software or hardware optimizations. Thus instruction set extension over an AMBA bus system did not offer any benefits and in contrast slowed down applications.

Another surprising result was the extremely fast scalar point multiplication by software (scheme I). It's 100 times faster than our hardware schemes. Cortex M1 has a internal SHIFT block (see **Figure 12**) which we assume is a very fast barrel-shifter. Also it provides a custom EOR instruction. Both these facts give it an advantage over other proprietary soft-cores. Shifts are costly in them and external IP to do these operations. The compiler in our case produces a much optimized code. The figures for our standalone software implementation are comparable to other soft-cores with custom hardware acceleration blocks added to them. Thus ARM Cortex M1 soft-core outperforms other soft-core plus custom accelerator configurations.

Earlier work	Clock freq MHz	Execution time $Q = k \cdot P$	Area overheads	Remarks
Picoblaze [35]	63	0.038 s	1841 LUT's	
Nios [17]	50	0.75 ms	(Mult+Div*=6589 LUT's)	Not given separately for multiplier
16 bit RISC [38]	5	0.5 s	NA	Multiplier modeled to work in 4 cycles
8051 [16]	12	99.2 ms	NA	Binary field 2^{163} field . 75 % extra area
Our design				

Cortex M1 + HW (scheme II and III)	20	Approx 0.5 – 0.6 s	866 core cells (actual 466 without APB wrapper)	32 cycle configurable dual multiplier
Cortex M1 + HW (scheme II and III)	50	Approx 0.23 s	866 core cells (actual 466 without APB wrapper)	32 cycle configurable dual multiplier
Cortex M1 (scheme I)	20	11-12 ms	NA	Particular binary 2^{163} field and fast reduction case

Table 6 Comparison of our work with similar approaches

***Mult+Div is combined area occupied by field multiplier and field divider**

If we include the configurable dual field multiplier in processor micro-architecture, the time for ECC scalar multiplication should be even less than 11 msecs at 20MHz. This is due to our expected speedups from section **3.4**. Thus combination of a high performing ARM architecture along with a custom ISE would give us best figures for such a hardware-software co-design platform. As shown in [48] , faster dual multipliers exist which in our case takes 32 cycles. The speed up in the multiplier itself, which is the main block of our algorithms, will highly speed up ECC.

5.3 Area and critical path

On synthesis, contribution of individual components to the total area can be shown by the following pie-chart. What interests us is the ratio of area of multiplier to area of Cortex M1 . The whole Multiplier along with the APB wrapper takes up 866 Tiles (in ACTEL terminology for FPGA LUT's) while Cortex M1 takes up 7491 tiles. Out of 866, only 496 is used for the multiplier with the rest being APB wrapper around it. Thus the multiplier is just **6.6 %** of total Cortex M1 area.

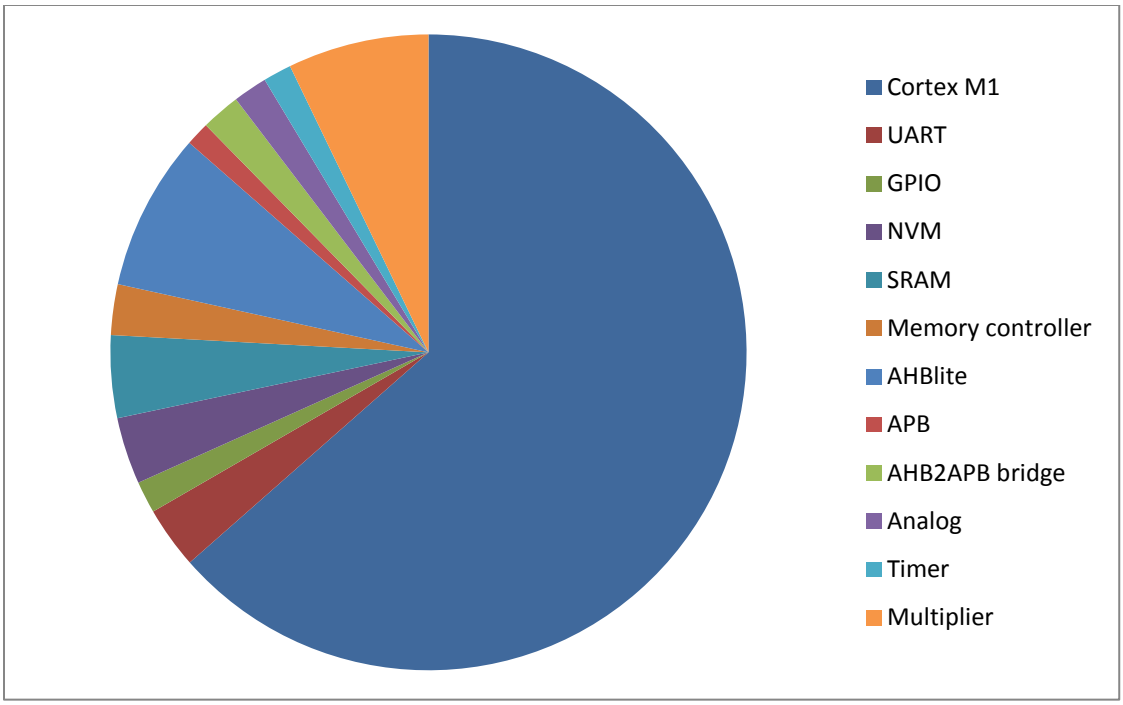


Figure 18 Area distribution of designed system

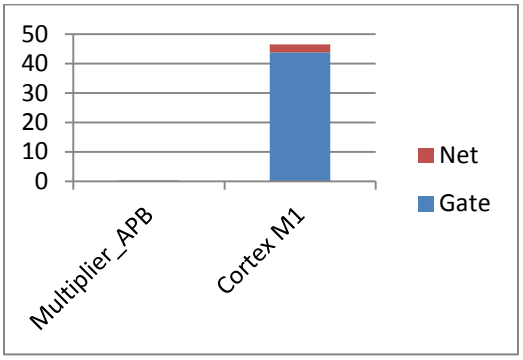
The Cortex is assumed to contain a similar shift and add integer multiplier in its data path because it is said to be a 33-cycle multiplier. Thus if our dual field multiplier replaces internal multiplier in the current configuration (028911 Refer Table 1 Page 6 [43]) the area overhead would be **negligible** consisting of only logic to make the same multiplier configurable. Also, as pointed out by [38] and [37] addition of this multiplier causes a very marginal increase in area and the critical path.

The multiplier when synthesized alone was able to work at maximum frequency of 61.25 MHz as compared to 61.74 MHz of Cortex M1 configuration used in the thesis. The timing analysis was done using SmartTime [47] on Actel M1AFS1500 130 nm LVCMOS process device. So, the addition of this configurable multiplier in place of already existing multiplier would not be a clock frequency limiting factor. It is to be noted that the frequency of operation could be much higher on a smaller process or on a faster ASIC.

5.4 Power Overheads

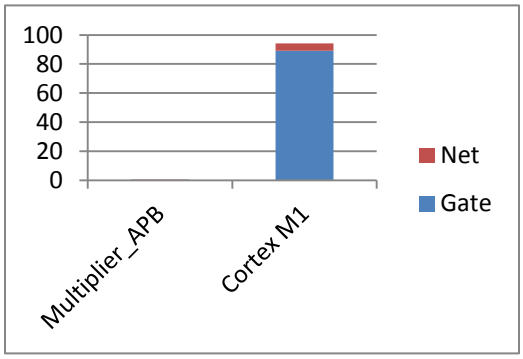
ACTEL provides a tool called as SmartPower [45] which can be used to measure static and dynamic power consumption of a design and even its components. It can analyze power consumption of IO, memories, nets, gates, peak power and also switching transients. We can specify the data switching frequency at the nets of the hardware module and it gives an estimate of power consumption. Also by default it could take 10 % data toggle rate.

Now, the processor was left to be at default 10 % toggle rate and we specified data frequencies at pins of our APB_multiplier on FPGA fabric. The number of MULGF2 operations in a multiplication is known from **Algorithm 6** and **Algorithm 7**. We also know the number of cycle's i.e. time to finish it. The figure comes to 81 KHz or 81000 MULGF2 accesses in a second for 20MHz system clock. An interesting observation was that the multiplier and Cortex M1 CPU as a part of the complete system showed **0 mW** of static power dissipation. The dynamic power dissipation of Cortex M1 and hardware multiplier is shown by the following graph



At 20 MHz Power in mW on Y-axis

Power overhead = 0.296/46.436 = 0.6 %



At 50 MHz Power in mW on Y-axis

Power overhead = 0.453/94.212 = 0.48 %

The figures might look quite optimistic due to assumptions made for calculations. The data frequencies at the multiplier were specified taking into account number of times it would be accessed per second, while Cortex M1 was assumed to have 10 % data switching frequency on its nets. This behavior can also be attributed to the low power nature of underlying Flash based FPGA fabric.

Chapter 6 Evaluation

We now review the aims and objectives declared at the outset of the project in the light of the approach undertaken and results in hand. Also, we can make sound conclusions based on the work carried out.

1. Comprehensive study of typical applications with algorithms to implement them

We undertook a detailed study of Elliptic curve cryptography (ECC) and FFT (Fast Fourier transform). The combination was inspired by use case of 'Wireless sensor networks' which are bunch of networked small signal monitoring units. Also worth mentioning is that, these applications were selected as their underlying behavior bore commonalities to many other applications (ECC with Error control coding and CRC, FFT with Discrete Cosine Transform and thus JPEG). Time constraints did not allow us to delve into more applications, but sufficient study of chosen two applications helped us take informed decisions about their implementation techniques considering trade-off's involved. Examples include choice to implement Binary finite field multiplication by 'Right to Left Comb method' and not 'Shift-and-add' method in software and choice of Projective co-ordinates in ECC over affine to trade-off for intensive binary finite field inversion. We were successful in identification of computation intensive tasks in these applications (as detailed in **section 3.2**) and worked on our objective to isolate the common ones in both of them. What we later did was actually devise a strategy to accelerate one of them (ECC) without any harm to the other.

2. Study of existing ARM ISA features

We had a fascination for ARM architecture and our objective was take into account existing micro-architectural strength's of ARM and propose other general purpose ISE to it. This is clearly evident by our detailed study on ARM ISA in **chapter 2** where we list out the instructions interesting for the applications in hand. We did an analysis of how these instructions already aid in fast software execution.

3. Effective hardware-software partitioning for ISE

As a continuation to the above two objectives we did an effective hardware software partitioning of the system we were building , best depicted by **Figure 7 and Figure 15**. We leveraged the SHIFT and XOR properties of ARM ISA and in holistic view of applications decided to do a MULGF2 i.e. polynomial multiplier extension. Addition and Reduction were decided to be done in software because of obvious support already provided by ARM ISE. We also researched for algorithms tweaked to use this ISE for better performance. We implemented 'Pencil-and-paper' and 'Comba's method' to perform binary finite field multiplication which used MULGF2. Similarly, squaring over the field was also done using MULGF2. At this stage, we predicted gains by use of MULGF2 to our binary finite field operation algorithms. We decided on this extension because the same integer multiplier

inside the core could be modified with minimum overheads to support this operation. So even though the FFT performance would not be affected (as we showed later no compromise on maximum operating frequency has to be done) ECC could be accelerated due to acceleration of these underlying field operations. So at this stage, after we had implemented FFT in software to profile it , we did not pursue of any further HW-SW co-design strategy for FFT. We settled for general purpose nature of our configurable dual field multiplier to perform both polynomial and integer multiplication.

4. To implement hardware peripheral and modify software to access the hardware module

We implemented hardware design of configurable dual field multiplier after designing the backbone system of Cortex M1, SRAM, Flash, UART's, GPIO, AMBA bus structure as detailed in **section 4.4**. The whole FPGA design flow of HDL, simulation, synthesis, place and route, Bus functional simulation and timing analysis was followed. Then we used **Algorithms 6,7** and **8** to target the configurable multiplier using device driver . The software development was done in SoftConsole. ARM Cortex M1 being an encrypted IP, does not allow modification to internal micro-architecture. So this peripheral was attached to APB bus. The architecture change for ISE in the microprocessor ALU data path would have given us true gain figures, but it was not possible. We had to evaluate a decoupled ISE. One thing worth a mention is that a faster multiplier designs [48], symbolic to fast multipliers in current processors could have been tried. But in the time available, it was not possible. Our multiplier still allowed us to measure tradeoffs and gains involved in the process.

5. Measurement of performance gain, code size change, area and power overheads of ISE to Cortex M1

As detailed in chapter 5, all the above factors were measured and the behavior was analyzed. The area and power overheads were negligible while code size remained almost same. The strategy to calculate power (**see section 5.4**) is approximate and nothing more can be done by tools available for the platform. Another way to calculate near- accurate values in such a hardware software partition is 'hybrid functional level power analysis (FLPA) and Instruction level power analysis (ILPA)' as in [49]. It allows to model power consumption of software tasks in a soft-core considering instruction level power dissipation. To do this for Cortex M1, could be a project in itself on our platform using modeling tools.

Anticipated gains were not obtained due to loosely coupled configuration of our hardware accelerator. It is not only necessary to develop fast hardware modules but considerable effort also has to be invested in communication strategies for them with host processors. Our hardware-software co-design strategy was almost 100 times slower than pure software implementation. Thus we could view practical aspects of hardware-software co-design in an existing industrial platform. But our strategy to evaluate ISE with a 32 bit Dual field multiplier was correct because it helped us measure actual costs and overheads

involved in the process. Our hardware peripheral if incorporated in processor micro-architecture can easily outperform all ECC designs in similar platforms (**Table 6**). This argument was detailed in **section 3.4**. Further, with a very well researched ECC software implementation we achieved one of the fastest figures to perform point multiplication. It easily outperforms other proprietary soft-cores in the market and a very good replacement to existing 8 and 16 bit RISC platforms. Even though it executed THUMB2 instruction set, it gave very good figures and a platform such as Cortex M3 with full ARM7TDMI instruction set would prove very powerful in these scenarios.

Thus in summary some points again worth a mention are -

ARM soft-core Cortex M1 outperforms to the best of author's knowledge, other available soft-core platforms. The powerful instruction set architecture allows it to proceed beyond the command-and-control function of a soft-core processor. As evident from the project, it can carry out complex software functions without much effect on performance.

While designing a hardware software system in a loosely coupled configuration, communication methods between host CPU and hardware peripheral need to be well researched, in some cases as good as the research on acceleration peripheral.

One of the efficient ways to do a hardware software co-design is instruction set extension but support in the form of tools needs to be available to actually harness the potential.

Chapter 7 Future Work

The immediate work that could be carried out would be to implement a dual field Wallace multiplier [48] and measure all parameters similar to our approach. As the authors argue on a 0.13 µm CMOS ASIC, it can work at more than 500 MHz it could well be included in modern processor's data path. The implementation of these multipliers over faster AHB bus could give us one of the fastest ECC systems. Besides, a complete ECC implementation could also be developed for e.g protocol such as ECDSA which would give us the exact figure of ECC computations.

On chip communication overhead in loosely coupled system is critical and efficient strategies to nullify their effect have to be investigated. In [15], the authors suggest use of coprocessor registers and local control signals to have some gains over conventional bus transfer protocol.

To support our multi-application approach for Instruction set extension, more applications could be considered. We could incorporate function like DCT due to similarities with FFT and do a re-check on hardware-software partitioning. Another strategy could be to analyze a benchmark set such as EEMBC [50] for high computation tasks. It may give us a range of correlations and help us make decisions about computation tasks that can be cheaply offloaded to hardware. It may be straightforward to find out computation blocks amongst functions in same suite say automotive suite or digital entertainment, but it would be challenging to build hardware blocks which could help in accelerating programs across a wide variety of suites. It needs to be seen, if we could with multiple iterations of this design cycle can actually propose methodologies for automated instruction set extension.

Bibliography

[1] F. Bensaali, A. Amira, and A. Bouridane, "An FPGA based coprocessor for 3D affine transformations," in *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, 15-17 Dec. 2003, pp. 288- 29.

[2] S.B Furber, *ARM System-on-Chip Architecture*.: Addison Wesley Longman, 2000.

[3] (2011, September) ARM Corporation Web site. [Online]. http://www.arm.com/products/processors/cortex-m/cortex-m1.php

[4] J.G. Tong, I.D.L. Anderson, and M.A.S. Khalid, "Soft-Core Processors for Embedded Systems," in *Microelectronics, 2006. ICM '06. International Conference on*, 16-19 Dec. 2006, pp. 170-173.

[5] Björn Stelte, "Toward development of high secure sensor network nodes using an FPGA-based architecture," in *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, 2010, pp. 539-543.

[6] M. de Miguel de Santos, C. Sanchez-Avila, and R. Sanchez-Reillo, "Elliptic curve cryptography on constraint environments," in *Security Technology, 2004. 38th Annual 2004 International Carnahan Conference on* , 11-14 Oct. 2004, pp. 212- 220.

[7] Marin L, Jara A, Skarmeta A.F.G. Ayuso J, "Optimisation of Public key Cryptography (RSA and ECC) for 16-bit Devices based on 6LoWPAN," in *1st international workshop on the Security of Internet of Things,Tokyo,Japan*, Tokyo,Japan.

[8] R. Furness, M. Benaissa, and S.T.J. Fenn, "GF(2m) multiplication over triangular basis for design of Reed-Solomon codes," in *Computers and Digital Techniques, IEE Proceedings* -, vol. 145, no.6, Nov 1998, pp. 437-443.

[9] C. Dreyer. The use of FFT's and other algorithms for fast Elliptic Curve operations. ECE 679 SPRING TERM JOURNAL OF CRYPTOGRAPHY.

[10] Hankerson D, Menezes A, and Vanstone S, *Guide to Elliptic Curve Cryptography*.: Springer-Verlag New York Inc, 2004.

[11] V Miller, "Use of elliptic curves in cryptography," *Lecture Notes in Computer Science*, vol. 218, pp. 417-426.

[12] Koblitz N, "Elliptic curve cryptosystems," *MATHEMATICS OF COMPUTATION*, 1987.

[13] Andrew Odlyzko, "Discrete Logarithms: The Past and the Future.," *Designs, Codes and Cryptography*, vol. 19, no. 2, pp. 129-145, Mar. 2000.

[14] Sandeep S. Kumar, "ELLIPTIC CURVE CRYPTOGRAPHY FOR CONSTRAINED DEVICES," Ruhr-University Bochum, Bochum, Germany, PhD Thesis June 2006.

[15] Xu Guo and Patrick Schaumont, "Optimizing the HW/SW boundary of an ECC SoC design using control hierarchy and distributed storage," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Nice, France, 2009, pp. 454-459.

[16] M Koschuch et al., "Hardware/Software Co-design of Elliptic Curve Cryptography on an 8051 Microcontroller," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, L Goubin and M Matsui, Eds.: Springer Berlin / Heidelberg, 2006, vol. 4249, pp. 430-444.

[17] Jian-Yang Zhou, Xiao-Wei Liu, and Xiao-Gang Jiang, "Implementing elliptic curve cryptography on Nios II processor," in *ASICON '07. 7th International Conference on*, 22-25 Oct. 2007, pp. 205-208.

[18] Julio López and Ricardo Dahab, "Fast Multiplication on Elliptic Curves over GF(2 ^m) without precomputation," in *Cryptographic Hardware and Embedded Systems*, Çetin Koç and Christof Paar, Eds.: Springer Berlin / Heidelberg, 1999, vol. 1717, p. 724.

[19] Lejla Batina, Sıddıka Berna Örs, Bart Preneel, and Joos Vandewalle, "Hardware architectures for public key cryptography," *Integration, the VLSI Journal*, vol. 34, no. 1-2, pp. 1-64, May 2003.

[20] J.W. Tukey and J.W. Cooley, "An algorithm for machine calculation of complex Fourier series," in *Math. Comp.*, vol. 55, 1965, pp. 297-301.

[21] (2011, September) ReliSoft Website. [Online]. http://www.relisoft.com/Science/Physics/fft.html

[22] A.C Elster, "Fast bit-reversal algorithms," in *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*, 23-26 May 1989, pp. 1099-1102.

[23] AMBA™ Specification (Rev 2.0), May 13, 1999, http://www.arm.com.

[24] (2011, September) [Online]. http://www.arm.com/products/processors/technologies/instruction-set-architectures.php

[25] L. Goudge and S. Segars, "Thumb: reducing the cost of 32-bit RISC performance in portable and consumer applications," *Technologies for the Information Superhighway*, pp. 176-181, Feb 1996.

[26] A. Kalavade and Subrahmanyam, "Hardware/software partitioning for multifunction systems," in *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, Sep 1998, pp. 819-837.

[27] D.R.H. Calderon Rocabado, "Arithmetic soft-core accelerators," TU Delft, Delft University of Technology, Delft, PhD Thesis 2007-11-27.

[28] J. Joven et al., "HW-SW implementation of a decoupled FPU for ARM-based Cortex-M1 SoCs in FPGAs," in *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, 15-17 June 2011, pp. 1-8.

[29] Chang Shu, Soonhak Kwon, and Kris Gaj, "FPGA accelerated tate pairing based cryptosystems over binary fields," in *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, Dec. 2006, pp. 173-180.

[30] Chu Chad, Zhang Qin, Xie Yingke, and Han Chengde, "Design of a high performance FFT processor based on FPGA," in *Proceedings of the ASP-DAC 2005. Asia and South Pacific*, vol. 2, Jan. 2005, pp. 920- 923.

[31] R. Singleton, "An algorithm for computing the mixed radix fast Fourier transform," in *Audio and Electroacoustics, IEEE Transactions on*, vol. 17, Jun 1969, pp. 93- 103.

[32] S. Morozov, C. Tergino, and P. Schaumont, "System integration of Elliptic Curve Cryptography on an OMAP platform," in *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, 5-6 June 2011, pp. 52-57.

[33] R. Meyer and K. Schwarz, "FFT implementation on DSP-chips-theory and practice," in *Acoustics, Speech, and Signal Processing, 1990. ICASSP-90., 1990 International Conference on* , vol. 3, 3-6 Apr 1990, pp. 1503-1506.

[34] Guidi F and Tortoli P Ricci S. (2011, September) Texas Instruments Web site. [Online]. http://focus-webapps.ti.com/pdfs/univ/02-ControlFilterDesignEstimation.pdf

[35] M.N. Hassan and Benaissa, "Embedded Software Design of Scalable Low-Area Elliptic-Curve Cryptography," in *Embedded Systems Letters, IEEE*, Aug. 2009, pp. 42-45.

[36] M. Frigo and S.G. Johnson, "The Design and Implementation of FFTW3," in *Proceedings of the IEEE*, vol. 93, no.2, Feb. 2005, pp. 216-231.

[37] Erkay Savas, Alexandre F. Tenca, and Çetin Kaya Koç, "A Scalable and Unified Multiplier Architecture for Finite Fields GF(p) and GF(2m)," in *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, 2000, pp. 277-292.

[38] J. Groszschaedl and G.-A Kamendje, "Instruction set extension for fast elliptic curve cryptography over binary finite fields GF(2m)," in *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, 24-26 June 2003, pp. 455- 468.

[39] (2011) Libero IDE v9.1 user guide. Document.

[40] A.M. Fiskiran and R.B Lee, "Evaluating instruction set extensions for fast arithmetic on binary finite fields," in *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, Sept. 2004, pp. 125- 136.

[41] Henry Gordon Dietz. (2011, September) The Aggregate Magic Algorithms. [Online]. http://aggregate.org/MAGIC/

[42] Eric W Weisstein. (2011, September) MathWorld--A Wolfram Web Resource. [Online]. http://mathworld.wolfram.com/ComplexMultiplication.html

[43] (2010, September) http://www.actel.com/documents. [Online]. http://www.actel.com/documents/CortexM1_HB.pdf

[44] (2011, September) ACTEL corporation Web site. [Online]. http://www.actel.com/documents/libero_ug.pdf

[45] (2011, September) Actel Corporation Web site. [Online]. http://www.actel.com/documents/smartpower_ug.pdf

[46] (2011, September) Actel Corporation Web site. [Online]. http://www.actel.com/documents/SoftConsole_UG.pdf

[47] (2010, November) SmartTime v9.1. Document. [Online]. http://www.actel.com/documents/smarttime_ug.pdf

[48] A. Satoh and K Takano, "A scalable dual-field elliptic curve cryptographic processor," in *Computers, IEEE Transactions on*, vol. 52, no.4, April 2003, pp. 449- 460.

[49] P Zipf et al., "A Power Estimation Model for an FPGA-Based Softcore Processor," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 27-29 Aug. 2007, pp. 171-176.

[50] J Poovey, M Levy, S Gal-On, and T Conte, "A Benchmark Characterization of the EEMBC Benchmark Suite," in *Micro, IEEE*, 2009, p. 1.

[51] A. de la Piedra, A. Touhafi, and G. Cornetta, "An IEEE 802.15.4 baseband SoC for tracking applications in the medical environment based on Actel Cortex-M1 soft-core,"

in *Communications and Vehicular Technology in the Benelux (SCVT), 2010 17th IEEE Symposium on*, 24-25 Nov. 2010, pp. 1-5.

[52] (2011, September) Aeroflex Gaisler Web site.

[53] M. Aydos, T. Yanik, and C.K. Koc, "High-speed implementation of an ECC-based wireless authentication protocol on an ARM microprocessor," in *Communications, IEE Proceedings-*, vol. 148, no.5, Oct 2001, pp. 273-279.

[54] Kumar S and Paar C, "Reconfigurable Instruction Set Extension for Enabling ECC on an 8-Bit Processor," in *International Conference on Field Programmable Logic and Application*, August 2004, pp. 586-595.

APPENDIX

```
Filename : Adder.v
module Adder(
             sin,
              pin,
             cin,
              fsel,
              cout,
             sout
              );

input  sin;
input  pin;
input  cin;
input  fsel;
output  cout;
output  sout;

wire nandg2,xnorg,nandg3,org;

assign nandg3 = ~(fsel&cin&sin);
assign xnorg = ~(cin^sin);
assign nandg2 = ~(pin&fsel);
assign org =  (xnorg | nandg2 );

assign cout = ~(org & nandg3);
assign sout = ~(xnorg ^ pin);

endmodule
```

```
Filename : Multiplier.v

module Multiplier( A,
                   B,
                  Clk,
                  RstN,
                   fsel,
                  Load,
                  Out,
                  Val
);

input  [31:0] A;
input  [31:0] B;
input  Clk;
input  RstN;
input  fsel;
input  Load;
output  [63:0]Out;
output  Val;

reg  [63:0] Hold;
reg  [5:0] Count;
reg  [31:0] TmpA;
//reg  [31:0] TmpB;

reg  Val;
wire [31:0] result;
wire [31:0] carry;

always @(posedge Clk or negedge RstN) begin //{
   if(!RstN)
     TmpA <= 32'h00000000;
   else if (Load)
     TmpA <= A;
end//}
```

```
//always @(posedge Clk or negedge RstN) begin //{
 //  if(!RstN)
  //   TmpB <= 32'h00000000;
   //else if (Load)
  //   TmpB <= B;
//end//}

always @(posedge Clk or negedge RstN) begin //{
   if(!RstN)
     Count <= 6'h00;
   else if (Load)
     Count <= 6'h20;
   else if (Count==6'h00)
     Count <= 6'h00;
   else
     Count <= Count - 1'b1;
end//}


always @(posedge Clk or negedge RstN) begin //{
   if(!RstN)
     Hold <= 64'd0;
   else if (Load)
     Hold <= {32'h00000000,B};
   else if(Count != 5'h00) begin //{
     if(Hold[0]) begin
        Hold <= {carry[31],result,Hold[31:1]};
     end //}
     else
        Hold <= {1'b0,Hold[63:1]};
   end //}
end//}

always @(posedge Clk or negedge RstN) begin //{
   if(!RstN)
     Val <= 1'b0;
   else if (Count == 5'h01)
     Val <= 1'b1;
   else
     Val <= 1'b0;
end//}

/*always @(posedge Clk or negedge RstN) begin //{
   if(!RstN)
     Out <= 64'd0;
   else if(Count == 5'h00) begin //{
     if(Hold[0]) begin
        Out <= {carry[31],result,Hold[31:1]};
     end //}
     else
        Out <= {1'b0,Hold[63:1]};
   end //}
end//}*/


assign Out = Hold;

// Adder

Adder bit0 (
               .sin(TmpA[0]),
               .pin(Hold[32]),
               .cin(1'b0),
               .fsel(fsel),
               .cout(carry[0]),
               .sout(result[0])
               );

Adder bit1 (
               .sin(TmpA[1]),
               .pin(Hold[33]),
```

```
                .cin(carry[0]),
                .fsel(fsel),
                .cout(carry[1]),
                .sout(result[1])
                );

Adder bit2 (
                .sin(TmpA[2]),
                .pin(Hold[34]),
                .cin(carry[1]),
                .fsel(fsel),
                .cout(carry[2]),
                .sout(result[2])
                );

Adder bit3 (
                .sin(TmpA[3]),
                .pin(Hold[35]),
                .cin(carry[2]),
                .fsel(fsel),
                .cout(carry[3]),
                .sout(result[3])
                );

Adder bit4 (
                .sin(TmpA[4]),
                .pin(Hold[36]),
                .cin(carry[3]),
                .fsel(fsel),
                .cout(carry[4]),
                .sout(result[4])
                );

Adder bit5 (
                .sin(TmpA[5]),
                .pin(Hold[37]),
                .cin(carry[4]),
                .fsel(fsel),
                .cout(carry[5]),
                .sout(result[5])
                );

Adder bit6 (
                .sin(TmpA[6]),
                .pin(Hold[38]),
                .cin(carry[5]),
                .fsel(fsel),
                .cout(carry[6]),
                .sout(result[6])
                );

Adder bit7 (
                .sin(TmpA[7]),
                .pin(Hold[39]),
                .cin(carry[6]),
                .fsel(fsel),
                .cout(carry[7]),
                .sout(result[7])
                );

Adder bit8 (
                .sin(TmpA[8]),
                .pin(Hold[40]),
                .cin(carry[7]),
                .fsel(fsel),
                .cout(carry[8]),
                .sout(result[8])
                );

Adder bit9 (
                .sin(TmpA[9]),
```

```
                .pin(Hold[41]),
                .cin(carry[8]),
                .fsel(fsel),
                .cout(carry[9]),
                .sout(result[9])
                );

Adder bit10 (
                .sin(TmpA[10]),
                .pin(Hold[42]),
                .cin(carry[9]),
                .fsel(fsel),
                .cout(carry[10]),
                .sout(result[10])
                );

Adder bit11 (
                .sin(TmpA[11]),
                .pin(Hold[43]),
                .cin(carry[10]),
                .fsel(fsel),
                .cout(carry[11]),
                .sout(result[11])
                );

Adder bit12 (
                .sin(TmpA[12]),
                .pin(Hold[44]),
                .cin(carry[11]),
                .fsel(fsel),
                .cout(carry[12]),
                .sout(result[12])
                );

Adder bit13 (
                .sin(TmpA[13]),
                .pin(Hold[45]),
                .cin(carry[12]),
                .fsel(fsel),
                .cout(carry[13]),
                .sout(result[13])
                );

Adder bit14 (
                .sin(TmpA[14]),
                .pin(Hold[46]),
                .cin(carry[13]),
                .fsel(fsel),
                .cout(carry[14]),
                .sout(result[14])
                );

Adder bit15 (
                .sin(TmpA[15]),
                .pin(Hold[47]),
                .cin(carry[14]),
                .fsel(fsel),
                .cout(carry[15]),
                .sout(result[15])
                );

Adder bit16 (
                .sin(TmpA[16]),
                .pin(Hold[48]),
                .cin(carry[15]),
                .fsel(fsel),
                .cout(carry[16]),
                .sout(result[16])
                );

Adder bit17 (
```

```
                .sin(TmpA[17]),
                .pin(Hold[49]),
                .cin(carry[16]),
                .fsel(fsel),
                .cout(carry[17]),
                .sout(result[17])
                );

Adder bit18 (
                .sin(TmpA[18]),
                .pin(Hold[50]),
                .cin(carry[17]),
                .fsel(fsel),
                .cout(carry[18]),
                .sout(result[18])
                );

Adder bit19 (
                .sin(TmpA[19]),
                .pin(Hold[51]),
                .cin(carry[18]),
                .fsel(fsel),
                .cout(carry[19]),
                .sout(result[19])
                );

Adder bit20 (
                .sin(TmpA[20]),
                .pin(Hold[52]),
                .cin(carry[19]),
                .fsel(fsel),
                .cout(carry[20]),
                .sout(result[20])
                );

Adder bit21 (
                .sin(TmpA[21]),
                .pin(Hold[53]),
                .cin(carry[20]),
                .fsel(fsel),
                .cout(carry[21]),
                .sout(result[21])
                );

Adder bit22 (
                .sin(TmpA[22]),
                .pin(Hold[54]),
                .cin(carry[21]),
                .fsel(fsel),
                .cout(carry[22]),
                .sout(result[22])
                );

Adder bit23 (
                .sin(TmpA[23]),
                .pin(Hold[55]),
                .cin(carry[22]),
                .fsel(fsel),
                .cout(carry[23]),
                .sout(result[23])
                );

Adder bit24 (
                .sin(TmpA[24]),
                .pin(Hold[56]),
                .cin(carry[23]),
                .fsel(fsel),
                .cout(carry[24]),
                .sout(result[24])
                );
```

```
Adder bit25 (
               .sin(TmpA[25]),
               .pin(Hold[57]),
               .cin(carry[24]),
               .fsel(fsel),
               .cout(carry[25]),
               .sout(result[25])
               );

Adder bit26 (
               .sin(TmpA[26]),
               .pin(Hold[58]),
               .cin(carry[25]),
               .fsel(fsel),
               .cout(carry[26]),
               .sout(result[26])
               );

Adder bit27 (
               .sin(TmpA[27]),
               .pin(Hold[59]),
               .cin(carry[26]),
               .fsel(fsel),
               .cout(carry[27]),
               .sout(result[27])
               );

Adder bit28 (
               .sin(TmpA[28]),
               .pin(Hold[60]),
               .cin(carry[27]),
               .fsel(fsel),
               .cout(carry[28]),
               .sout(result[28])
               );

Adder bit29 (
               .sin(TmpA[29]),
               .pin(Hold[61]),
               .cin(carry[28]),
               .fsel(fsel),
               .cout(carry[29]),
               .sout(result[29])
               );

Adder bit30 (
               .sin(TmpA[30]),
               .pin(Hold[62]),
               .cin(carry[29]),
               .fsel(fsel),
               .cout(carry[30]),
               .sout(result[30])
               );

Adder bit31 (
               .sin(TmpA[31]),
               .pin(Hold[63]),
               .cin(carry[30]),
               .fsel(fsel),
               .cout(carry[31]),
               .sout(result[31])
               );



endmodule
```

```
Filename : Multiplier_APB.v
// Multiplier_APB.v
module Multiplier_APB
```

```
(
PCLK,
PRESETN,
PADDR,
PSEL,
PENABLE,
PWRITE,
PRDATA,
PWDATA,
PREADY,
PSLVERR
);


input PCLK;
input PRESETN;
input [23:0] PADDR;
input PSEL;
input PENABLE;
input PWRITE;
output PREADY;
output PSLVERR;
output [31 :0] PRDATA;
input  [31: 0] PWDATA;

wire wr_enable,rd_enable;

reg [31:0] regA;
reg [31:0] regB;
reg [31:0] regC;
reg [63:0] Result;
//reg [31:0] data_out;
reg [31:0] PRDATA;

wire [63:0] temp_Result;
wire [31:0] hold2 ;
wire [31:0] hold1 ;
wire val1;

assign PREADY = 1'b1;
assign PSLVERR = 1'b0;

assign wr_enable = (PENABLE &  PWRITE & PSEL);
assign rd_enable = (!PWRITE & PSEL);

assign hold2 = Result[63:32];
assign hold1 = Result[31:0];
assign noval=32'b00000000;

parameter FirstOperandAddress = 24'h000000;
parameter SecondOperandAddress = 24'h000004;
parameter ControlRegisterAddress = 24'h000008;
parameter ResultLowerWordAddress = 24'h00000C;
parameter ResultHigherWordAddress = 24'h000010;


always @(posedge PCLK or negedge PRESETN) begin //{

    if(!PRESETN)
        regA <= 32'h00000000;

    else if(PADDR == FirstOperandAddress & wr_enable)

        regA <= PWDATA;
    end //}

always @(posedge PCLK or negedge PRESETN) begin //{

    if(!PRESETN)
        regB <= 32'h00000000;
```

```
    else if(PADDR == SecondOperandAddress & wr_enable)

        regB<= PWDATA;
    end //}

always @(posedge PCLK or negedge PRESETN) begin //{

    if(!PRESETN)
        regC <= 32'h00000000;

    else if(PADDR == ControlRegisterAddress & wr_enable)

        regC<= PWDATA;

    else if (val1)

            regC[2] <= 1'b1;

    else regC[1] <= 1'b0;

    end //}

always @(posedge PCLK or negedge PRESETN) begin //{

    if(!PRESETN)
        Result <= 64'h00000000;

    else if(val1)

        Result<= temp_Result;

    end //}

/*

always @(posedge PCLK or negedge PRESETN) begin //{

    if(!PRESETN)
        data_out <= noval;

    else if(PADDR == ControlRegisterAddress & rd_enable)

        data_out<= regC;

    end //}


always @(posedge PCLK or negedge PRESETN) begin //{

    if(!PRESETN)
        data_out <= noval;

    else if(PADDR == ResultLowerWordAddress & rd_enable)

        data_out <= hold1;


    end //}

always @(posedge PCLK or negedge PRESETN) begin //{

    if(!PRESETN)
        data_out <= noval;

    else if(PADDR == ResultHigherWordAddress & rd_enable)

        data_out <= hold2;


    end //}
```

```
always @(posedge PCLK or negedge PRESETN) begin //{

    if(!PRESETN)

        PRDATA <= 32'h00000000;

    else if(rd_enable)

        PRDATA <= data_out;


    end //} */


always @(posedge PCLK or negedge PRESETN) begin //{

    if(!PRESETN)

        PRDATA <= 32'h00000000;

    else if(rd_enable)

        case (PADDR)

     ControlRegisterAddress : PRDATA <= regC;

     ResultLowerWordAddress : PRDATA <= hold1;

     ResultHigherWordAddress :PRDATA <= hold2;

        default :             PRDATA <= 32'h00000000;

        endcase

    end //}


    Multiplier m (
                 .A(regA),
                 .B(regB),
                 .Clk(PCLK),
                 .RstN(PRESETN),
                 .fsel(regC[0]),
                 .Load(regC[1]),
                 .Out(temp_Result),
                 .Val(val1)
                 );


endmodule
```

SOFTWARE code submitted as a part of zip file.