DESIGN AND REALIZATION OF AN EMBEDDED PROCESSOR FOR CRYPTOGRAPHIC APPLICATIONS

by

ÖVÜNÇ KOCABAŞ

Submitted to the Graduate School of Engineering and Natural Sciences

in partial fulfillment of

the requirements for the degree of

Master of Science

Sabancı University

August, 2008

DESIGN AND REALIZATION OF AN EMBEDDED PROCESSOR FOR CRYPTOGRAPHIC APPLICATIONS

APPROVED BY

Assoc.Prof.Dr. Erkay Savaş	....
(Dissertation Supervisor)	
Assoc.Prof.Dr. Albert Levi	....
Assist. Prof.Dr. İlker Hamzaoğlu	....
Assist. Prof.Dr. Selim Balcısoy	....
Assist. Prof.Dr. Yücel Saygın	....

DATE OF APPROVAL: ....

© Övünç Kocabaş 2008

All Rights Reserved

DESIGN AND REALIZATION OF AN EMBEDDED PROCESSOR

FOR CRYPTOGRAPHIC APPLICATIONS

Övünç KOCABAŞ

CS, Master of Science Thesis, 2008

Thesis Supervisor: Assoc. Prof. Dr. Erkay Savaş

Keywords: embedded processors, public key cryptography, architectural enhancements,symmetric key cryptography, cache based attacks

Abstract

Architectural enhancements are a set of modifications in a general-purpose processor to improve the processing of a given workload such as multimedia applications and cryptographic operations. Employing faster/enhanced arithmetic units for the existing instruction set architecture (ISA), introducing application-specific instructions to the ISA, and adding a new set of registers are common practices employed as architectural enhancements.

In this thesis, we introduce and implement a set of relatively low-cost enhancement techniques to accelerate certain arithmetic operations common in cryptographic applications on a configurable and extensible embedded processor core. The proposed enhancements are generic in the sense that they can profitably be applied in many RISC processors. These enhancements are organized into, what we prefer to call as, cryptographic unit (CU) that offers an extended ISA to the programmer. We then present the speedup values obtained for various arithmetic operations and public key cryptography

algorithms through these enhancements. Furthermore, hardware overhead of introducing the enhancements to the embedded extensible processor is provided in terms of chip area. Our experimental results show that the proposed architectural enhancements provides significant amount of speedup (up to one order of magnitude) in elliptic curve cryptography and RSA with a conservative increase in hardware. Last but not the least, we demonstrate that the proposed enhancements facilitate protection of cryptographic algorithms against certain side-channel attacks by reporting our case study of AES implementation hardened against cache-based attacks.

KRİPTOGRAFİK UYGULAMALAR İÇİN

GÖMÜLÜ İŞLEMCİ TASARIMI VE UYGULAMASI

Övünç KOCABAŞ

CS, Master Tezi, 2008

Tez Danışmanı: Doç. Dr. Erkay Savaş

Anathar kelimeler: gömülü işlemciler, açık anahtarlı şifreleme, mimari geliştirmeler, gizli anahtarlı şifreleme, önbellek temelli ataklar

Özet

Mimari iyileştirmeler, genel amaçlı işlemcilerin çoğul ortam uygulaması ve kritografik işlemler gibi işyüklerindeki performansını arttırmak icin yapılan değişikliklerdir. Varolan komut kümesi mimarisi için yeni ve geliştirilmiş aritmetik birimler kullanmak, komut kümesi mimarisine yeni uygulamaya özgü işlemler tanıtmak ve yeni yazmaç kümesi eklemek genel olarak kullanılan mimari iyileştirme teknikleridir.

Bu tezde, kriptografik uygulamalarda kullanılan aritmetik işlemleri hızlandırmak amacıyla nispeten düşük maliyetli iyileştirme teknikleri önerilmiş ve bu tekniklerin uygulaması yapılmıştır. İyileştirme teknikleri çoğu RISC işlemcisine uygulanabilecek şekilde tasarlanmıştır. Bu iyileştirmeler Kriptografik Birim olarak organize edilmiş ve programcıya genişletilmiş komut kümesi mimarisi olarak sunulmuştur. Öngörülen iyileştirmeler kullanıldığında çeşitli aritmetik işlemler ve açık anahtarlı şifreleme algoritmaları için hızlanma değerleri sunulmuştur. Ayrıca, genişletilebilir gömülü mimariler için

önerilen iyileştirmelerin uygulanması sonucunda oluşan donanım gideri yonga alanı olarak gösterilmiştir. Yapılan deneyler sonucunda önerilen iyileştirmeler sayesinde eliptik eğri şifreleme ve RSA sistemlerinde makul bir donanım artışı karşılığında önemli seviye de hızlanma kaydedilmiştir. Son olarak önerilen iyileştirmelerin aynı zamanda kriptograpfik algoritmaların bazı yan kanal ataklarına karşı korunmasında yardımcı olacağı gösterilmiştir.

Acknowledgements

First and foremost, I wish to express my gratitude to my thesis supervisor Erkay SAVAŞ for his valuable advice and guidance during my thesis study. His complementary knowledge on cryptography and digital system design was inspirational during my research and I am grateful to him not only for the completion of this thesis, but also his patience and unconditional support.

I am grateful to my thesis committee members Albert LEVİ, İlker HAMZAOĞLU, Selim BALCISOY and Yücel SAYGIN for their valuable review and comments on my master thesis.

Furthermore, I would like to thank The Scientific and Technological Research Council of Turkey (TÜBİTAK) for their financial support during my graduate study so that I can concentrate my research and complete my thesis.

Last but not least, I would like to thank my family for always being there for me, supporting my decisions and encouraging me throughout my graduate education.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Background Information 3

1.2.1 Public Key Cryptography 3

1.2.2 RSA 4

1.2.3 Elliptic Curve Cryptography (ECC) 6

1.3 Previous Works and Motivation 9

1.4 Contribution 10

1.5 Organization of the Thesis 11

2 General Architecture **14**

2.1 Configurable Processors 14

2.2 Tensilica Xtensa Processor Cores 14

2.2.1 LX2 Cores 15

2.3 Generating Custom *cryptographically-enhanced processor* . . . 16

2.3.1 Base Processor 17

2.3.2 Building *cryptographically-enhanced processor* 19

2.3.3 *Cryptographic Register File (CRF)* 20

2.3.4 *Cryptographic Execution Unit (CEU)* 21

2.3.5 Integer Unit 22

2.3.6 Multiply Unit 24

2.4 128-bit Multiplication Implementation Details 24

2.4.1 Computing Partial Products 26

2.4.2 Alignment and Addition of Partial Products 27

2.5 Proposed Instructions 29
2.6 Total Hardware Cost 31

3 Modular Multiplication **33**
3.1 Montgomery Multiplication 33
3.1.1 Methods for Montgomery Multiplication 35
3.1.2 The Separated Operand Scanning (SOS) Method . . . 36
3.1.3 The Coarsely Integrated Operand Scanning (CIOS) Method 39
3.1.4 Enhanced SOS Method 41
3.1.5 Performance Analysis 43

4 Modular Inversion **45**
4.1 Modular Inversion in finite $GF(p)$ 45
4.1.1 Kaliski and Montgomery Inversion Algorithm 46
4.1.2 Implementation Details 51
4.1.3 Performance Analysis 51

5 Implementation Details **54**
5.1 FPGA Emulation and Time-Area Metrics 56

6 An AES Implementation Hardened Against Cache Attacks **59**

7 Conclusion and Future Work **62**

List of Figures

1 Point Addition Operation on elliptic curves 7
2 Point Doubling Operation on elliptic curves 8
3 Xtensa LX2 Core 16
4 General Architecture of Enhanced Embedded Core 20
5 Detailed Architecture of the CU 22
6 128-bit carry select adder 23
7 Dividing 128-bit multiplication into four 64-bit multiplication 25
8 Computing Final Product 25
9 Partial Product Computation 27
10 Alignment and Addition of Partial Sums 28

List of Tables

1 Configuration of *base processor* 18

2 List of Instructions 31

3 Hardware Cost of *CU* (5 stage pipeline) 32

4 Hardware Cost of *CU* (7 stage pipeline) 32

5 Speedups for Modular Multiplication on 5-stage pipeline version 44

6 Speedups for Modular Multiplication on 7-stage pipeline version 44

7 Montgomery Inversion on *base processor* 52

8 Montgomery Inversion on *cryptographically-enhanced processor* 52

9 Kaliski Inversion on *base processor* 52

10 Kaliski Inversion on *cryptographically-enhanced processor* . . . 52

11 Montgomery Inversion Speedups 53

12 Kaliski Inversion Speedups 53

13 Implementation Results for Elliptic Curve Point Multiplication 54

14 $Time \times Area$ product for RSA 57

15 $Time \times Area$ for ECC 57

16 Improvements for RSA and ECC 58

17 Overhead of protecting rounds of AES in number of clock cycles 61

List of Algorithms

1 Binary Exponentiation Algorithm 6

2 Montgomery Multiplication 34

3 Separated Operand Scanning (SOS) Method 38

4 Coarsely Integrated Operand Scanning (CIOS) method 40

5 Enhanced SOS Method 42

6 Kaliski Inversion Algorithm 49

7 Montgomery Inversion Algorithm 50

1 Introduction

1.1 Introduction

When embedded microprocessors made their first presence a few decades ago, they were merely low-end micro-controllers designed to perform only simple control instructions [9]. Ever since with the escalating innovations in integrated circuit technology, the role of embedded microprocessors is also revolutionized. Nowadays embedded microprocessors are used in almost every aspects of daily life, ranging from portable devices to large stationary installations. Furthermore, complexity of these processors rises up from single low-end micro-controller unit to multiple units integrated into one board with peripherals and network connection.

ARM, MIPS and Power PC are some of the examples of the most widespread embedded microprocessor architectures which were developed in the 1980's for stand-alone microprocessor chips. These architectures are excelled in performing wide range of algorithms. However with the emergence of innovative research areas and their applications fields, such as multimedia and communication applications, more processing power is demanded by designers. Public key cryptosystems, which employ multi-precision arithmetic, also require more processing power since overwhelming majority of their running time is spent in a few performance-critical sections. A common solution for the related performance problem is two-fold: either designers move on to a processor which has a higher clock frequency or they can design custom hardware for boosting up the performance of the critical portions of their design. Former is the most straightforward yet old-fashioned method, where the in-

creasing clock frequency triggers excessive power consumption which turns out to be yet another problem for the designers. In the latter method, designers build custom hardware blocks by using hardware description languages (e.g. VHDL and Verilog) to speed-up the hot spots of their applications. This method is extensively used for reaching high frequency values which embedded microprocessors fail to respond. However, most of the time designing a custom RTL hardware consumes significant amount of time and effort. Verifying the RTL hardware takes even more time and once designed, these hardware blocks cannot be changed easily. Due to these issues, RTL hardware design for performance enhancement may become complicated task for the designers.

A novel solution for boosting up performance is to use configurable processors instead of embedded microprocessors and RTL hardware blocks for specific applications that demand high performance. These processors are a new family of processor cores, in which one can modify a processor for a specific application. These cores are much faster, more efficient and able to perform more than standard embedded microprocessors.

This work explores the benefits of architectural enhancements for fast and secure computation of cryptographic operations on a configurable processor. The enhancements come in three flavors: 1) configuring processor core, 2) extending architecture with new functional units with reasonable overhead and 3) augmenting the existing ISA with new instructions. The performance of public key cryptography is primarily determined by the efficient implementation of arithmetic operations in the underlying algebraic structure (e.g. finite field). Extending a general purpose processor through relatively low-cost en-

hancement techniques for fast arithmetic operations, which dominate cryptographic computations in terms of time and resource usage, has a number of benefits over using hardware accelerator such as a cryptographic co-processor which is in the category of RTL design. First, performing the cryptographic operations within processor core eliminates the communication overhead and possibly associated security risks, accrued in processor/co-processor settings. Second, the area of a cryptographic co-processor is generally much larger than the area overhead of proposed enhancements that are tightly coupled to the processor core and directly exploited by the instruction stream. Third, architectural enhancements offer a degree of flexibility and scalability that goes far beyond of fixed-function hardware such as a co-processor since extended architecture still be used for general-purpose computing with the potential benefit for other application domains as well.

1.2 Background Information

In this section we elaborate on two public key cryptography schemes e.g. RSA and Elliptic Curve Cryptography which are implemented on enhanced processor.

1.2.1 Public Key Cryptography

Public Key Cryptography, which is also named as asymmetric cryptography, is proposed as a solution to distribution and management of secret keys. In a network environment with n users, $n(n-1)/2$ keys should be generated and distributed and implementing this structure without using a secure channel is a difficult problem. The first solution to the problem was introduced by

Diffie and Hellman [8] in 1976.

In public key cryptography, every user has a pair of keys: public key and private (secret) key. The private key is only known to user while public key can be distributed to the network. A generic public key cryptography protocol between two users, Alice and Bob, is as follows. First Bob sends his public key to Alice. Alice encrypts her message by using Bob's public key and sends encrypted message to Bob. Bob decrypts the encrypted message by using his private key. In this protocol, only Bob can decrypt the message since only he knows the secret key. Both public and private key is related to each other mathematically but by knowing public key, private key cannot be derived in practical computation limits.

1.2.2 RSA

RSA is the most widely known and used public key cryptography algorithm. It is invented by Rivest, Shamir and Adleman in 1978 [25]. In RSA, each user has private and public key pair. The private key of the user in RSA system is consists of two large primes, p and q, and a secret exponent d. The public key of the user is $n = p \cdot q$ and e with the properties

$$e = d^{-1} \mod \Phi(n)$$

$$gcd(e, \Phi(n)) = 1$$

where $\Phi(n)$ is Euler's Totient Function and $\Phi(n) = (p-1) \cdot (q-1)$.

In a RSA setting, sender encrypts the message m by using receiver's public key e and sends the encrypted message $c = m^e \mod n$ to the receiver. To

decrypt the encrypted message, receiver uses his private key and compute the following

$$m = c^d = m^{e \cdot d} = m^{1+k\Phi(n)} = m \mod n$$

Decryption can be performed as shown above according to Fermat's Little Theorem. Fermat's Little Theorem states that an integer a and prime number p has the relationship of

$$a^{p-1} = 1 \mod p$$

Fermat's Little Theorem can be generalized as Euler's Totient Function as follows

$$a^{\Phi(p)} = 1 \mod p$$

where a and p are relatively prime to each other.

The most important operation in RSA is the modular exponentiation operation. But the numbers used in RSA are big integers, for a minimum level of security 1024-bit secret keys must be used, therefore it will take long time to perform modular exponentiation if it is performed as successive modular multiplications. Instead Binary Exponentiation Algorithm (c.f. Algorithm 1) is used to speedup the modular exponentiation.

Algorithm 1 Binary Exponentiation Algorithm

Input: m is the base, e is k-bit exponent in binary form $(e_{k-1}, e_{k-2}, ...e_1, e_0)$
Output: $product = m^e$

1. $product = 1$
2. **for** $i = k - 1$ **to** $i = 0$
3. $product = product \times product$
4. **if** $(e_i = 1)$**then** $product = product \times m$
5. **return** $product$

1.2.3 Elliptic Curve Cryptography (ECC)

Neal Koblitz [17] and Victor Miller [21] independently proposed new standards for public key cryptography which is called as Elliptic Curve Cryptography(ECC). They showed that a group defined on an elliptic curve can be used for cryptographic operations. For cryptographic applications, elliptic curves defined on prime field *GF(p)* or binary extension field *GF(2^n)* can be chosen.

An elliptic curve over *GF(p)* is defined as the set of solutions to the following equation

$$y^2 = x^3 + a \cdot x + b$$

where a and b are elements in prime finite field. If a point (x, y) satisfies the above equation then it is on the elliptic curve. All points satisfy the equation above and the *infinity point*, which is denoted as θ, over prime finite field, form an additive group and point addition operation is the group operation.

The point addition of two points, $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, on the elliptic curve is as follows

$$R = P + Q = (x_3, y_3)$$

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \mod p$$

$$x_3 = \lambda^2 - (x_1 + x_2) \mod p$$

$$y_3 = (\lambda \cdot (x_1 - x_3) - y_1) \mod p$$

where λ is the slope of the line, passing through points P and Q. The point addition operation is presented in Figure 1.

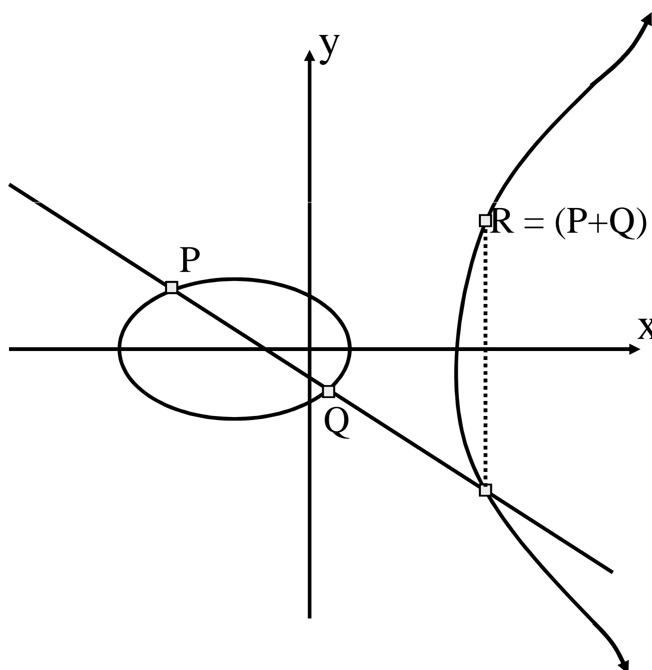

Figure 1: Point Addition Operation on elliptic curves

Another version of point addition is point doubling where $S = 2P$ is

computed as follows (c.f Figure 2).

$$S = 2P = (x_3, y_3)$$

$$\lambda = \frac{3 \cdot x_1^2 + a}{2 \cdot y_1} \mod p$$

$$x_3 = \lambda^2 - (2 \cdot x_1) \mod p$$

$$y_3 = (\lambda \cdot (x_1 - x_3) - y_1) \mod p$$

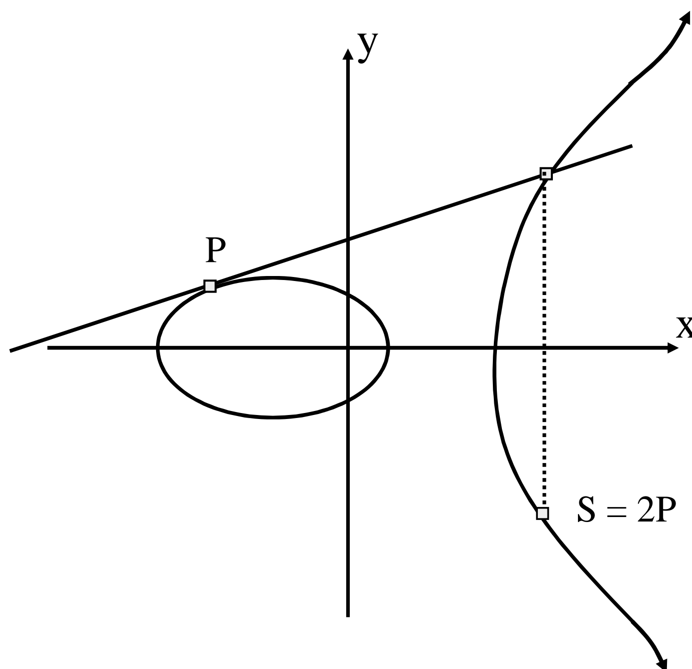

Figure 2: Point Doubling Operation on elliptic curves

The modular exponentiation operation of RSA is equivalent to point multiplication operation in ECC. In point multiplication, a point on the elliptic curve is multiplied with a scalar and the result of the multiplication resides again on the elliptic curve. Point multiplication operation is performed as repeated point addition and point doubling. The advantage of the ECC over RSA is that in ECC same security level of RSA can be achieved by using shorter key lengths. For instance, 1024-bit RSA security level is equivalent to 160-bit key length in ECC. This property makes ECC a promising PKC since the encryption operation can be performed faster than the RSA and shorter key lengths and digital signatures are required to RSA with the same level of security.

1.3 Previous Works and Motivation

Previous works [12, 13, 30, 31, 11] propose various enhancements to accelerate cryptographic operations. For instance, the authors in [12] propose five custom instructions to accelerate arithmetic operations in both $GF(p)$ and $GF(2^n)$ on MIPS32 core to benefit elliptic curve cryptography while ISA extensions in [31] aim to accelerate pairing-based cryptography. Similarly, the authors in [11] explore the effects of on-chip memory on the execution time of s-box computations in symmetric key cryptography. A common feature of these works is that they focus on custom solutions for accelerating an individual cryptographic operation on general-purpose processors.

In this work, we take a slightly different and holistic approach by designing and integrating so called *Cryptographic Unit (CU)* into a configurable and extensible processor core. Numerous cryptographic operations will benefit

from CU for fast and secure execution. The proposed CU facilitates new and powerful instructions and hardware extensions to accelerate multiplication and inversion in prime finite field $GF(p)$ and cryptographic operations which are performed in RSA and elliptic curve cryptography. It is also shown that CU is instrumental for software implementation of AES which is resistant to side-channel attacks.

1.4 Contribution

In public key cryptography, the most important operations are finite field arithmetic operations. In Diffie-Hellman key exchange [8], RSA [25] and digital signature systems [23] modular exponentiation is the most important and time consuming operation which is performed as repeated modular multiplications. Also for Elliptic Curve Cryptography (ECC), point multiplication operation is the most expensive operation in terms of time and area. Point multiplication operation is performed as point doubling and point addition operations. These operations consist of modular inversions, modular multiplications and modular additions. Thus overall performance of public key cryptosystems is determined by the performance of arithmetic operations in finite fields.

In this thesis, we proposed a *Cryptographic Unit (CU)* for fast and secure execution of the arithmetic operations in finite fields. The proposed CU is generic thus it can be integrated into many RISC based processors. Within the CU a cryptographic register file and a cryptographic execution unit are introduced. Besides, new instructions are defined to employ the units in the CU.

An enhanced processor is designed by integrating the *CU* on a configurable and extensible processor core. Arithmetic operations are implemented on the enhanced processor and the speedup values are up to 13.1 times for modular multiplication and 4.6 times for modular inversion. Both RSA and ECC operations are implemented on the enhanced processor as well and a performance improvement of 10.1 times for RSA and 8.08 times for ECC are obtained.

The enhanced processor is later mapped to a specific FPGA board (Avnet LX200) and hardware cost and clock frequency of the processor are obtained. The clock frequency of the processor demonstrates that the *CU* does not increase the critical path delay while introducing additional hardware to processor core. By using the implementation results, $time \times area$ product is computed for both RSA and ECC to investigate if the speedups are profitable. The $time \times area$ product shows that by employing the *CU* an improvement up to 6.64 times in RSA and 4.69 times in ECC can be achieved. The results prove that the benefits of the proposed *CU* far exceed its cost.

Finally, it is shown that using the *CU* can be instrumental for protecting software implementation of AES from certain side channel attacks (cache-based attacks) with a reasonable overhead in execution time.

1.5 Organization of the Thesis

The outline of the rest of thesis is as follows:

- Chapter 2 reveals the detailed architecture of custom processor designed for cryptographic applications. It starts with the designing process of the custom processor on configurable and extensible *base*

processor. Architectural enhancements and new set of instructions are introduced later. Finally hardware cost of implementing custom processor is provided in number of gates in $0.13\mu m$ technology.

- Chapter 3 explains Montgomery's method for modular multiplication. It discusses methods for implementing Montgomery Multiplication on hardware. Modified version of one of the discussed methods is presented which utilizes the enhanced architecture of custom processor. The chapter ends with the comparison of modified method for custom processor with the most efficient method for implementation on *base processor*.

- Chapter 4 starts with the definition of modular inversion operation in $GF(p)$ finite fields. It introduces two efficient algorithms for computing modular inverse in hardware. The chapter ends with the comparison of both algorithm's performance on custom processor and *base processor*.

- Chapter 5 shows the impact of the proposed enhancements presented in Chapter 3 on RSA and elliptic curve cryptography. The speedups for RSA and elliptic curve cryptography are presented. Implementation of the enhanced processor on specific FPGA board is explained and finally time $\times$ area products of RSA and elliptic curve cryptography on custom processor and *base processor* are compared.

- Chapter 6 moves to symmetric key cryptography with the focus on AES. A side channel attack e.g. cache based attack, against software implementation of AES is introduced. Counter measures to protect

software implementation of AES are discussed. Finally the overhead of protection mechanisms are presented in terms of execution time.

- Chapter 7 concludes the thesis and discusses on future work possibilities.

2 General Architecture

2.1 Configurable Processors

A typical configurable processor consists of a pre-defined processor core which can be enhanced for specific application requirements. Configuring these processor cores generally includes modifications, additions or removals to processor peripherals, memories, external bus widths and handshake protocols. One can add as many functional units as possible for performance improvement and still keep the area small by removing the unnecessary parts for the specific application. Once finished with the configuration, configurable processors are synthesized as RTL code and can be mapped to ASIC or FPGA's. ARC [3], Improv [14], Tensilica [27] are some of the major companies that offer configurable processor cores.

Tensilica's Xtensa configurable processor cores are preferred as the target embedded processor in our work, since they are one of the configurable cores that offer full software-development tool chain, including compiler, debugger and ISS (Instruction Set Simulator) to match the configured processor. In addition, the Tensilica Xtensa cores are also extensible; a property that make them a superset of configurable processors, offering more flexible solutions compared to the other configurable-only processors.

2.2 Tensilica Xtensa Processor Cores

Tensilica offers two types of Xtensa configurable cores: LX2 and Xtensa 7, which are intended for embedded applications. While Xtensa 7 is optimized for low power applications such as control operations, LX2 cores are more

flexible and ideal for high performance demanded data-incentive operations. Among these cores we choose LX2 cores for our base processor since we will be dealing with multi-precision arithmetic in finite fields and performing these operations will require more processing power.

2.2.1 LX2 Cores

Xtensa's LX2 32-bit processor architecture features a compact instruction set optimized for embedded system designs. The base architecture includes a 32-bit ALU, up to 64 general-purpose physical registers, 80 base instructions including 16 and 24-bit instruction encoding instead of RISC encoding which enables significant code size reductions [29]. Furthermore LX2 core has two essential features; namely configurability and extensibility, which will be utilized in the process of generating custom *cryptographically-enhanced processor*.

Configurability attribute of LX2 core offers designers to robust their design for the specific applications where they can modify the processor core according to their design specs. Modification of processor can be made by defining the width and number of execution units, data interfaces and optional data paths. Whereas with extensibility feature, custom execution units, registers, register files, single-instruction multiple-data functional units can be added to processor data path. Extensions to data path is achieved through Tensilica Instruction Extension (TIE) language. TIE is a Verilog-like language which is used to describe instruction set extensions to processor core. Functional behaviors of desired extensions are defined in TIE and TIE compiler will generate and place the RTL equivalent blocks into processor

data path. A typical LX2 processor core is given in Figure 3 [28].

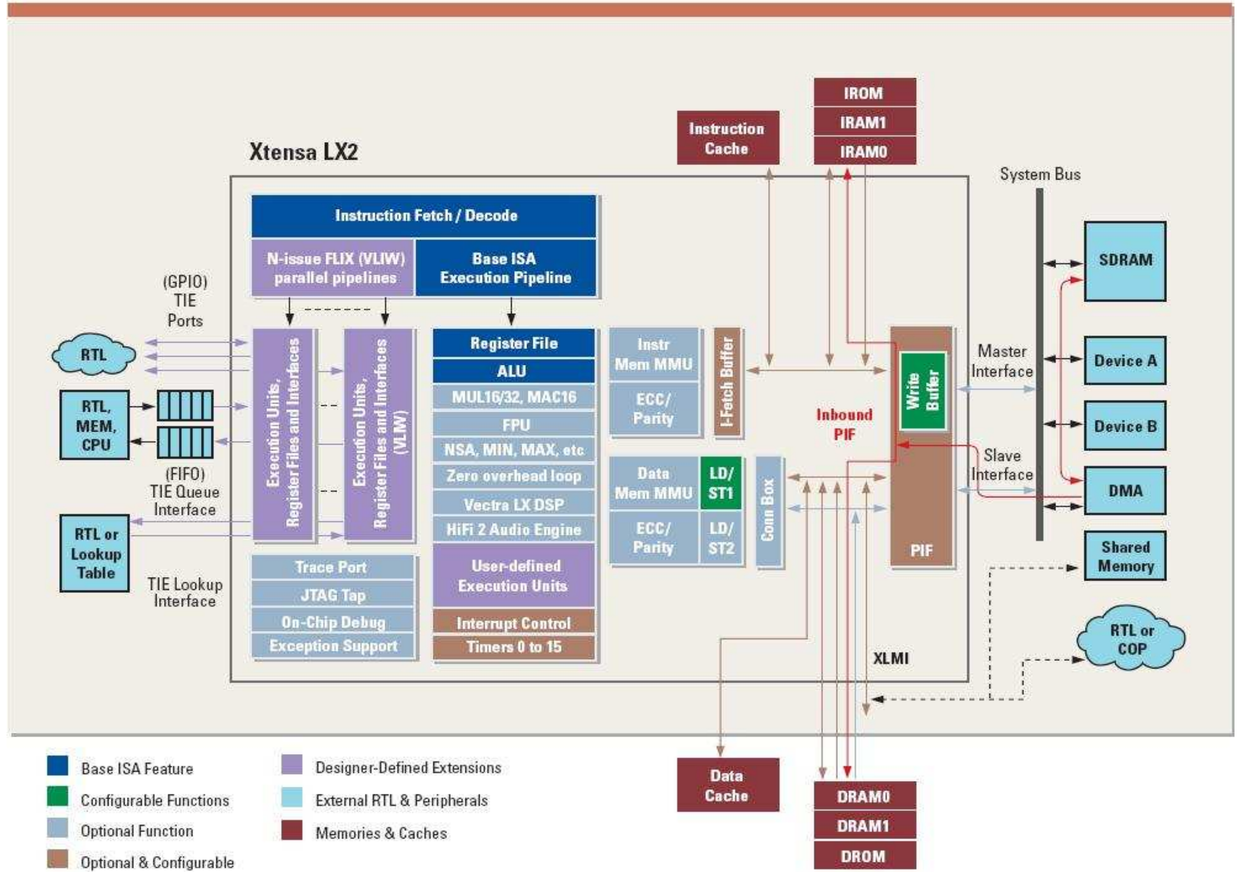

Figure 3: Xtensa LX2 Core

2.3 Generating Custom *cryptographically-enhanced processor*

Our design criteria for generating a custom *cryptographically-enhanced processor* is to build a processor which provides not only fast and secure execution of public key cryptography algorithms of RSA and elliptic curve cryptog-

raphy but also a core that is resistant to certain side-channel attacks against the software implementation of symmetric key cryptography algorithms, e.g. AES.

Design process of creating such processor consists of two steps. First, LX2 processor core is configured into so called *base processor* and then the *base processor* is extended with custom instructions and functional units by using TIE language to build final configuration which we name as *cryptographically-enhanced processor*.

Xtensa Xplorer Integrated Design Environment (IDE) is utilized during design and implementation steps of *cryptographically-enhanced processor*. Xplorer IDE tool integrates software development and processor optimization tools into one common environment and it provides all necessary tools for processor and TIE development, software development and modeling and simulation.

All the applications and the public key cryptography algorithms are developed in C programming language. In the performance analysis sections of the following chapters, arithmetic operations and public key cryptography algorithms are compared according to their execution times in terms of clock cycles. Clock cycle values are obtained by executing code on Xplorer IDE and looking the profile information, which is generated by the cycle-accurate Instruction Set Simulator (ISS) of the Xplorer IDE.

2.3.1 Base Processor

Cryptographically-enhanced processor is designed for embedded systems and configuration of *base processor* is performed depending on the requirements

of embedded systems. Therefore we aim to generate a processor as compact as possible yet still efficient enough to perform fast execution of cryptographic operations.

To keep processor size as small as possible, unnecessary units are removed from LX2 core. For instance, floating point unit is removed from core since public key operations are performed by using integer arithmetic. Also 32-bit integer divider is removed as the division operations in cryptographic algorithms will be performed by shifting the value to the right . Data and instruction caches are also chosen as reasonable sizes and direct-mapped cache.

To increase processor's performance, memory-cache interfaces and Processor Interface(PIF) are chosen as 128-bit (largest available) to increase bandwidth and word size of processor. The configuration of *base processor* is presented in Table 1.

Unit	Configuration
Multiply Unit	32 bit
Register File	32×32-bit
Data memory/cache interface	128-bit
PIF interface	128-bit
Data Cache	8KB / direct-mapped / 16byte line size
Instruction Cache	8KB / direct-mapped / 16byte line size

Table 1: Configuration of *base processor*

Pipeline length of the LX2 core is also configurable and two versions of *base processor* are generated with 5 and 7 stage pipeline length. The hardware cost of 5 and 7 stage pipelined versions of *base processor* in $0.13\mu m$ CMOS technology is as follows

- A total of approx. 119,000 gates with 5-stage pipeline configuration,

- A total of approx. 137,000 gates with 7-stage pipeline configuration.

2.3.2 Building *cryptographically-enhanced processor*

Prior to proposing architectural extensions and new instructions to *base processor*, following criteria are taken into consideration and enhancements are proposed in a way that they do not result in:

- unacceptable increase in area,
- change in instruction format and size,
- difficult integration with available tool-chain(e.g. compilers, debuggers, linkers),
- major change in the control circuitry and existing pipeline structure

Extensions to the *base processor* are done by integrating a new unit referred as *cryptographic unit (CU)* and introducing new set of instructions to core ISA. Figure 4 shows the *CU* which consists of two parts: *cryptographic register file (CRF)* and *cryptographic execution unit (CEU)*. In the following sections, the *CRF* and the *CEU* are explained in detail prior to introducing new instructions.

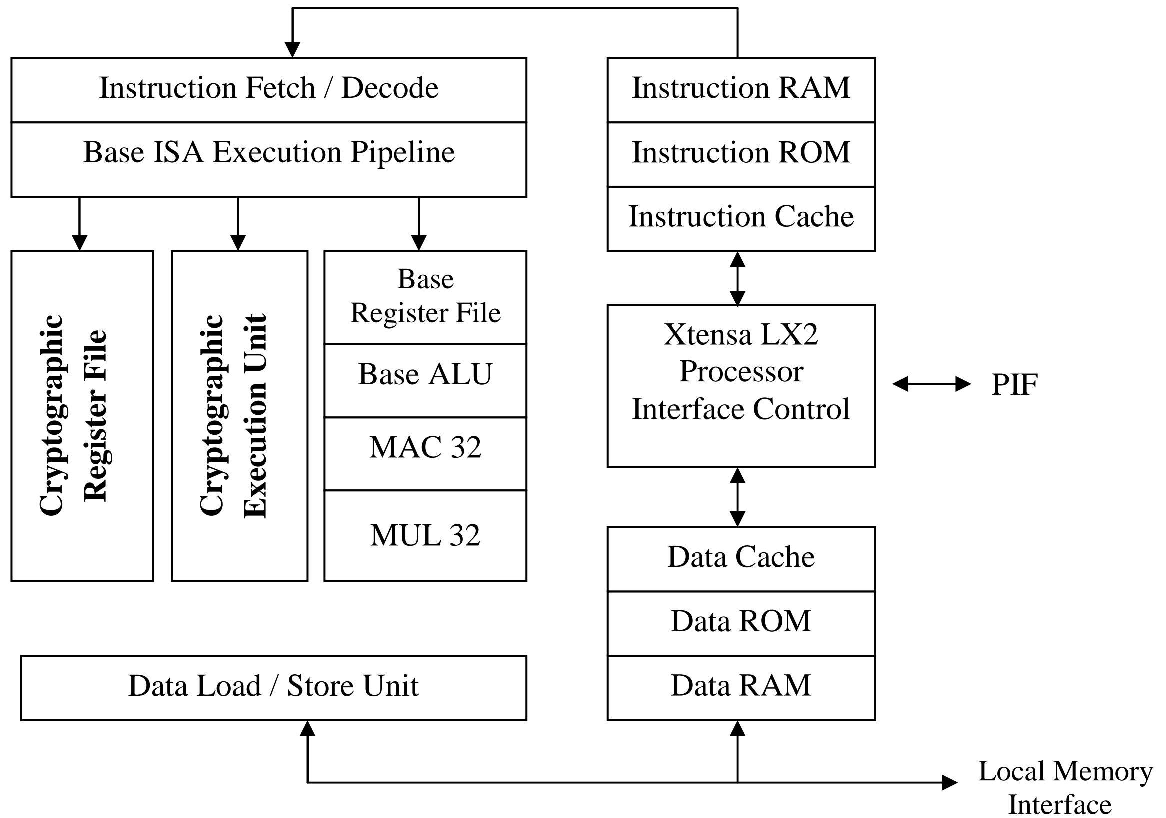

Figure 4: General Architecture of Enhanced Embedded Core

2.3.3 *Cryptographic Register File (CRF)*

The *CRF* is an array of 32 registers each of which has 128-bit width and is used to store operands and temporary results of arithmetical operations. Storing these values in the *CRF* will significantly reduce the execution time since the number of time consuming memory access operations will be reduced. Besides, the *CRF* can be used to store sensitive information such as secret keys and small look-up tables for increasing security level of cryptographic algorithms. In Chapter 6, we will show that the *CRF* will be

of crucial importance for protecting software implementation of AES from side-channel attacks; e.g. cache attacks.

Furthermore, the *CRF* can be shared by different processes if the operating system supports multi-tasking. In order to alleviate the security and switching cost concerns, we propose transactional usage of the *CRF*. The content of the *CRF* is not saved by the operating system on context switching; therefore any process that wants to use *CRF* does not automatically assume that the register contents remain intact forever. The process is provided with a consistent view of the *CRF* for only short duration (e.g. the duration of one multi-precision multiplication). It can lock the *CRF* for this duration so that no other process can use the *CRF* if the context switching occurs too frequently. The operating system can assist process for a fair schedule of the *CRF* usage in order to prevent starvation or attacks by malicious processes. A smart scheduling algorithm can easily solve the aforementioned problems.

2.3.4 *Cryptographic Execution Unit (CEU)*

The *CEU* is the new execution unit designed to utilize 128-bit width processor interface and the *CRF* during cryptographic operations. By choosing interface precision as 128-bit we simply increase our word length to 128-bit for cryptographic operations instead of 32-bit word size of general purpose processors. Using 32-bit ALU in the core processor will be inefficient for these operations therefore the *CEU* is designed to be used as functional unit for cryptographic operations. Functional units of the *CEU* will now take their operands from the *CRF* instead of 32-bit physical registers of core processor.

The *CEU* is composed of three parts: an integer unit, a shifter circuit and

a multiply unit. While *Integer Unit (IU)* is capable of adding/subtracting and comparison of two 128-bit integers, shifter circuit performs shift operation on both directions on a 128-bit register. Final functional unit in the *CEU* is multiply unit which performs 128-bit multiplication, and generates 256-bit result and stores the most and least significant 128 bit of the result on special purpose registers HI and LO respectively. Figure 5 shows the detailed architecture of the *CU* and functional units inside the *CEU*.

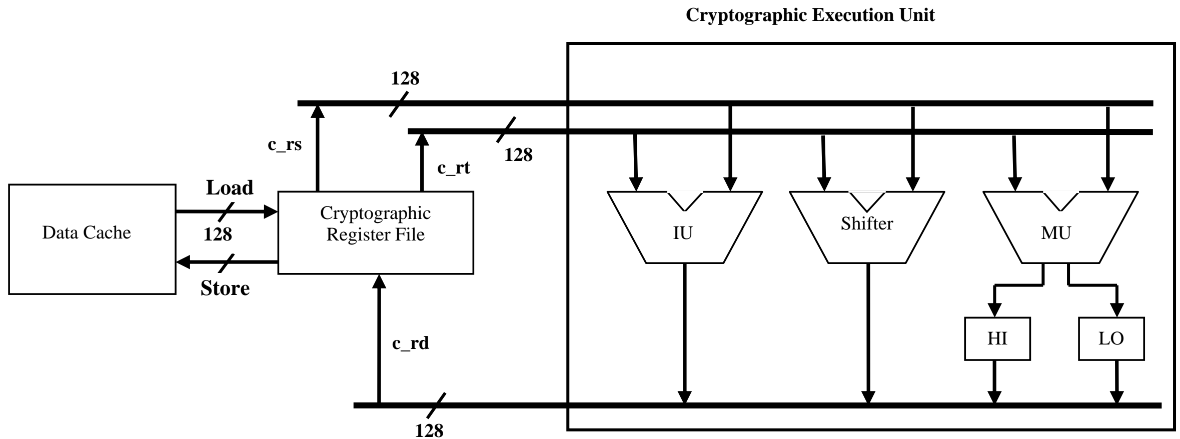

Figure 5: Detailed Architecture of the *CU*

2.3.5 Integer Unit

The *Integer Unit (IU)* consists of two parts: 128-bit adder and 128-bit comparator. While realization of the comparator is done straightforward, the adder in the *IU* is implemented as carry select adder. The carry select adder which is illustrated in Figure 6 consists of three 64-bit ripple carry adders and one multiplexer.

Carry select adder is preferred to 128-bit ripple carry adder since utilizing a 128-bit ripple carry adder will increase the critical path delay. By implementing carry ripple adder, latency of the 128-bit addition is reduced to 64-bit addition. Implementation of the carry select adder is performed by splitting 128-bit operands into 2 parts: 64-bit most significant part and 64-bit least significant part. First least significant 64-bit parts are added to each other and one bit carry is generated as a result. Meanwhile for the most significant part, two addition is computed one with the assumption of carry is being zero and the other with the carry is being one. The carry value generated from the least significant part of the addition is used for selecting the result from one of the additions performed for the most significant part.

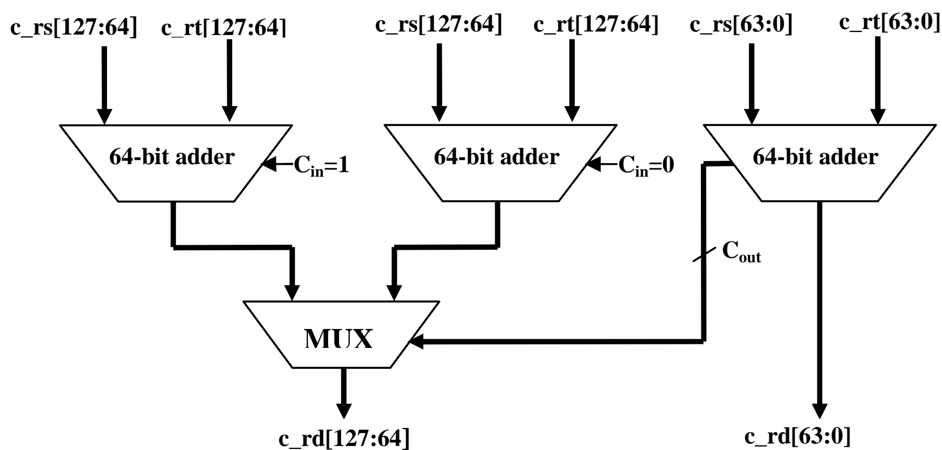

Figure 6: 128-bit carry select adder

2.3.6 Multiply Unit

Multiply unit is the most crucial functional unit of the *CEU* for accelerating modular multiplication operations which is excessively performed in RSA and elliptic curve cryptography. To speed up the multiplication operation we will utilize four parallelized 32-bit multipliers without increasing critical path delay(CPD). However, choice of the number of multipliers is critical due to their expensive cost in terms area and number of gates.

Performing a 128-bit multiplication requires 16 32-bit multiplications. One can choose to instantiate 16 multipliers to calculate all 32-bit multiplications in parallel and one cycle then add the partial products appropriately to get the final result. Yet using 16 32-bit multipliers will severely increase the processor area. Instead we prefer to implement 128-bit multiplication by utilizing four 64-bit multiplications and add the aligned partial products to get 256-bit result. In each 64-bit multiplication four 32-bit multiplications will be performed and we will utilize four 32-bit multipliers to execute them parallel. By using 4 parallel multipliers instead of 16 we will still get a significant speed up at the expense of acceptable hardware cost.

2.4 128-bit Multiplication Implementation Details

The proposed *cryptographically-enhanced processor* has 128-bit word size therefore, all multiplication operations are performed on 128-bit operands. Implementation of 128-bit multiplication will be performed as follows, first 128-bit multiplication will be divided into four 64-bit multiplications (Figure 7). Each 64-bit multiplication produces a partial product and in the end all

partial products will be aligned and added each other to compute final product. Final product, which is 256-bits, is stored on HI and LO special purpose registers as presented in Figure 8. First computation of partial products in parallel by using four multipliers will be explained and then alignment and addition of partial products into final product will be shown as successive iterations in Figure 8.

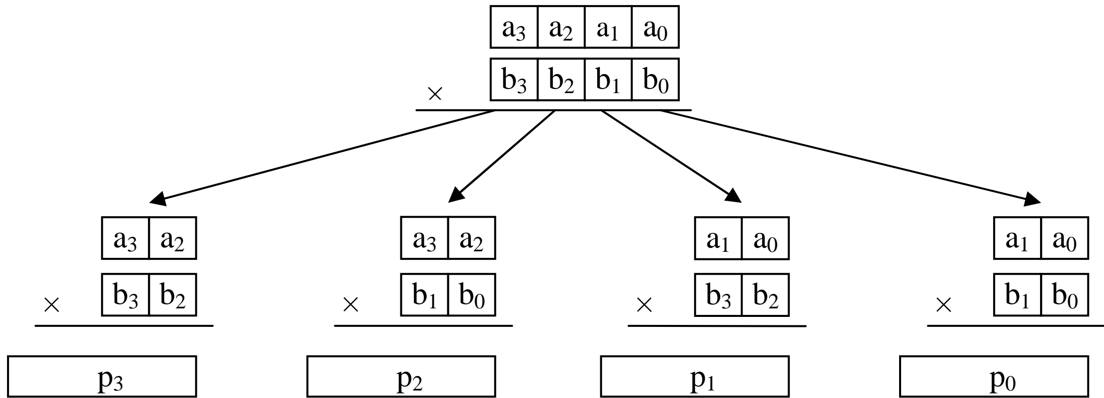

Figure 7: Dividing 128-bit multiplication into four 64-bit multiplication

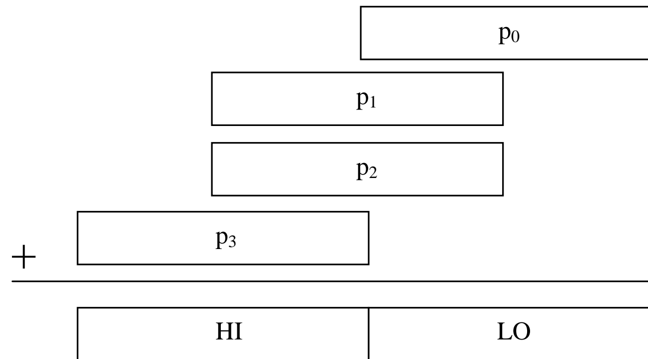

Figure 8: Computing Final Product

2.4.1 Computing Partial Products

In a 64-bit multiplication, four 32-bit multiplications are performed and with four multipliers in the multiply unit, these multiplications can be computed in parallel and in first clock cycle. HI register stores the t_l and t_h of the results while LO register stores t_{int1} and t_{int2}. Before calculating the partial product, which is 128 bits, two more operations have to be performed. First, the intermediate results are added (t_{int1} and t_{int2}) and then the sum is aligned and added to the value in HI register. After these operations the partial product is calculated and stored in a 128-bit register. Figure 9 shows the process of partial product calculation.

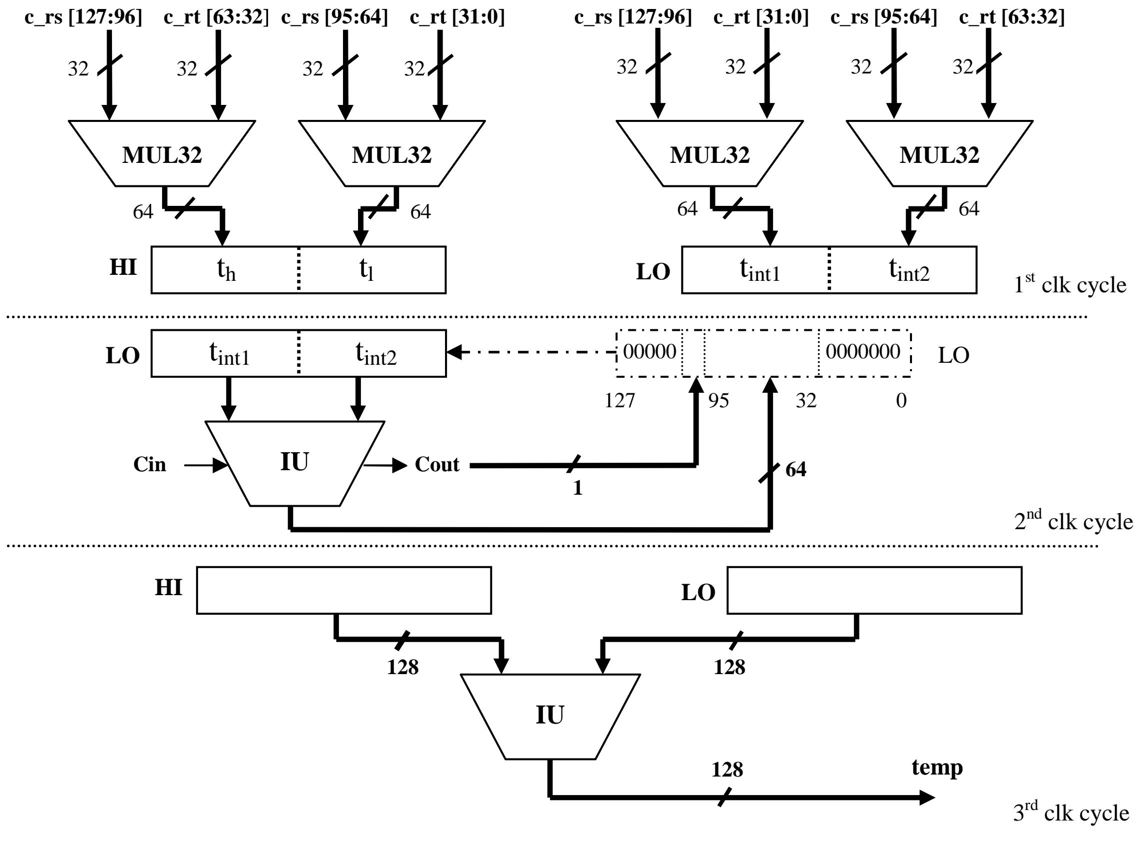

Figure 9: Partial Product Computation

2.4.2 Alignment and Addition of Partial Products

Four partial products of each 64-bit multiplications namely p_0, p_1, p_2, p_3 (cf. Figure 10) which are calculated in the previous step, are stored temporarily in four 128-bit registers. Final product is computed after three iterations which is composed of successive additions of partial products into HI and LO registers. These iterations are also summarized in Figure 10.

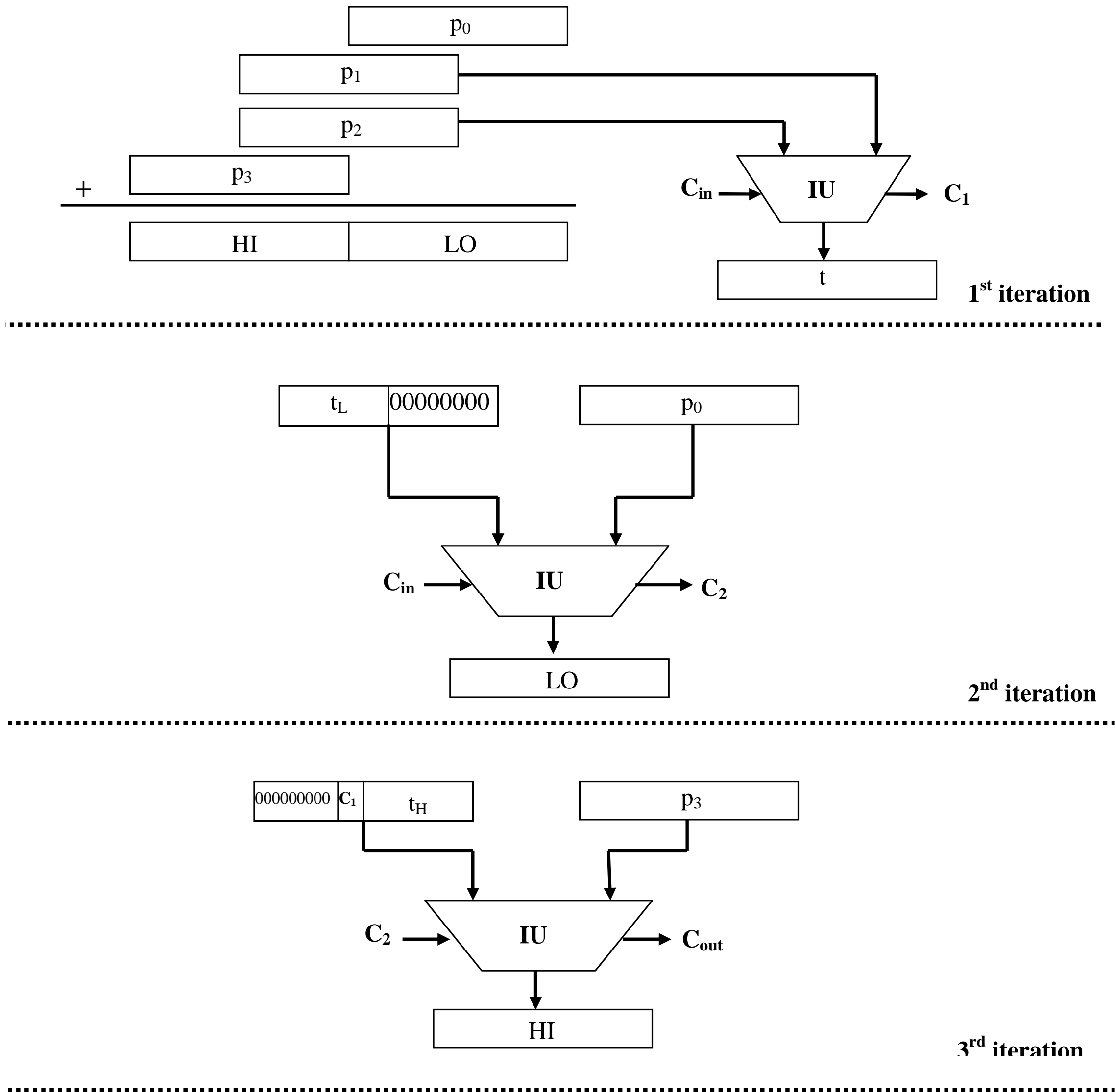

Figure 10: Alignment and Addition of Partial Sums

1st iteration: Partial products p_1 and p_2 are added and the result (t) is stored temporarily in a register (In following iterations, t will be divided into two halves, t_H and t_L, and each half will be used as operands of addition to HI and LO registers). Also the carry of the addition, C_1, is stored in one bit carry register as it is used in the calculation of result on HI register in the final iteration.

2nd iteration: In this iteration, lower half of the partial sum calculated in first iteration, t_L, is added with the p_0 and result will be the lower half of the final product and stored in the LO register. Again the carry out from this step, C_2, is stored in a carry register and is used in the final iteration.

3rd iteration: With the final step, final product is calculated and stored in HI and LO registers. In this iteration, upper half of the partial sum of the first iteration, t_H, is concatenated with C_1 and summed up with the p_3. During the addition, carry of second iteration, C_2, is used as carry-in value. Finally, result of the addition is stored in HI register.

2.5 Proposed Instructions

A new family of instructions is introduced to the processor ISA to fully employ the *CEU*. These instructions operate on 128-bit operands and conform to instruction type and formats of LX2 core which uses RISC instruction encoding. Therefore new instructions are encoded as RISC instructions with a slight difference. Common notations of source, target and destination registers (denoted as *rs*, *rt* and *rd* respectively) in RISC encoding are adjusted to reflect changes such that functional units in the *CEU* uses operands stored in the *CRF*. Therefore source, target and destination registers of the *CRF*

are represented as *c_rs*, *c_rt* and *c_rd*.

All proposed instructions are presented in Table 2. ADD_CREG and SUB_CREG operations perform unsigned addition and subtraction respectively. Both operations take their operands from *c_rs* and *c_rt* registers and write result back to *c_rd* register. COMP_CREG operation compares the values of *c_rs* and *c_rt* registers and if the value of *c_rs* register is greater than *c_rt* register than it writes 1 to *c_rd* otherwise it writes 0. SHL_CREG and SHR_CREG operations perform 1 bit shift operation. The *CRF* has two read ports and only one write port therefore the value of *c_rs* register can be changed while the value in *c_rt* register remains unchanged. MUL_CREG operation performs 128-bit unsigned multiplication and writes product to HI and LO special purpose registers. Finally, LOAD_CREG and STORE_CREG operations perform data transfer operations between memory and the *CRF* for given memory address.

Format	Description	Operation
ADD_CREG (c_rd,c_rs,c_rt)	Unsigned Addition	(C_{out} , c_rd) $\leftarrow$ c_rs + c_rt + C_{in}
SUB_CREG (c_rd,c_rs,c_rt)	Unsigned Subtraction	(B_{out} , c_rd) $\leftarrow$ c_rs - c_rt - B_{in}
COMP_CREG (c_rd,c_rs,c_rt)	Comparison	c_rd = c_rs > c_rt ? 1 : 0
SHL_CREG (c_rs ,c_rt)	Shift together left	c_rs $\leftarrow$ c_rs[126:0] || c_rt[127]
SHR_CREG (c_rs ,c_rt)	Shift together right	c_rs $\leftarrow$ c_rt[0] || c_rt[127:1]
MUL_CREG (c_rs,c_rt)	Unsigned Multiplication	(HI / LO) $\leftarrow$ c_rs $\times$ c_rt
LOAD_CREG (c_rd)	Load data from memory	c_rd $\leftarrow$ Memory [address]
STORE_CREG (c_rd)	Store data to memory	Memory [address] $\leftarrow$ c_rd

Table 2: List of Instructions

2.6 Total Hardware Cost

Introducing the *CU* to *base processor* increases the total area. The hardware costs of the units inside the *CU* are given in terms of gates in $0.13\mu m$ CMOS technology (c.f. Table 3 and 4) for both 5 and 7 stage pipeline versions of *base processor*. Cost of the *CRF* includes number of gates for 32×128 bit register file. Multiply unit's cost includes four 32-bit multipliers and four 128-bit registers which store the partial products during a 128-bit multiplication. While cost of the *IU* includes 128-bit adder and comparator circuit. The

rest of additional hardware cost including multiplexing and decoding circuit given under Other costs part in Tables 3 and 4.

Unit	**Gate Count**
base processor	118,475
CRF	46,631
Multiply Unit	42,471
IU	5,113
Shifter	35
Other	15,576
CU	109,946

Table 3: Hardware Cost of *CU* (5 stage pipeline)

Unit	**Gate Count**
base processor	136,829
CRF	48,452
Multiply Unit	46,236
IU	5,122
Shifter	35
Other	15,929
CU total	115,774

Table 4: Hardware Cost of *CU* (7 stage pipeline)

3 Modular Multiplication

3.1 Montgomery Multiplication

The Montgomery multiplication for fast computation of modular multiplication of big integers is proposed by P.L. Montgomery [22]. The Montgomery Multiplication algorithm (MM) computes the following:

$$MM(X, Y, N) = X \cdot Y \cdot R^{-1} \mod N \tag{1}$$

where X and Y are the multiplicand and multiplier respectively, N is the modulus and R is an integer with the property $gcd(N, R) = 1$. One can choose any R however if R is chosen as power of 2 (e.g. 2^k) then the implementation of Montgomery multiplication on microprocessors turns out to be very fast. While calculating

$$X \cdot Y \mod N$$

requires trial division by N, Montgomery multiplication only needs division by a power of two, $R = 2^k$, which can be performed by shifting result k times to right and shift operation is executed very fast in microprocessors and also comes with a low cost in software and free in hardware.

Prior to performing Montgomery multiplication, all operands need to be translated to their *N-residue* representation. *N-residue* of an integer a is denoted as

$$a_R = a \cdot R \mod N$$

where $R = 2^k$. The set of $\{a \cdot R \mod N \mid 0 \leq a \leq n-1\}$ is a complete residue system and includes all numbers between $[0, p-1]$. The numbers in the range $[0, p-1]$ have a one-to-one correspondence with residue set given above. Montgomery Multiplication employs the property of the residue system above and computes the *N-residue* product of two *N-residue* integers efficiently. Montgomery Multiplication consists of two steps as described in Algorithm 2. First multiplication of two residue numbers is calculated and then product is reduced to its final form. For the reduction step an additional quantity, N', is defined with the following property

$$R \cdot R^{-1} - N \cdot N' = 1$$

where both N' and R^{-1} can be calculated by using extended Euclidean algorithm.

Algorithm 2 Montgomery Multiplication

1: $T = a_R \cdot b_R$
2: $U = (T + (T \cdot N' \mod R) \cdot N)/R$
3: **if** $U \geq N$ then **return** $U - N$ **else return** U

The step 2 of Montgomery Multiplication algorithm involves modulo R and division by R operations. These operations are executed very fast in microprocessors since division by $R = 2^k$ means just shifting result right by k times and modulo R operation is performed by taking only lower k bits of product and discarding the rest.

The flow of operations for performing the Montgomery Multiplication given in Equation 1 are defined as follows

- Conversion of X to *N-residue* form

$$X_R = MM(X, R^2, N) = X \cdot R^2 \cdot R^{-1} = X \cdot R \mod N$$

- Conversion of Y to *N-residue* form

$$Y_R = MM(Y, R^2, N) = Y \cdot R^2 \cdot R^{-1} = Y \cdot R \mod N$$

- Computation of product in *N-residue* form

$$P_R = MM(X_R, Y_R, N) = X_R.Y_R.R^{-1} \mod N$$

$$P_R = X \cdot R \cdot Y \cdot R \cdot R^{-1} = X \cdot Y \mod N$$

- Conversion of the product from its *N-residue* form

$$P = MM(P_R, 1, N) = X.Y.R.1.R^{-1} = X.Y \mod N$$

To perform one Modular multiplication with Montgomery algorithm, four multiplications have to be calculated. Also for reduction an extra effort is made to compute value of N'. Therefore using Montgomery Multiplication for single modular multiplication is not feasible. Montgomery Multiplication become efficient when several modular multiplications have to be performed as in the case of modular exponentiation. In this case, the *N-residue* representation of intermediate results can be maintained while only conversion operation is needed during first and last multiplication.

3.1.1 Methods for Montgomery Multiplication

An overview of five different algorithms for Montgomery Multiplication is provided by Koç et al. [19]. Organization of these algorithms is based on two facts:

- whether multiplication and reduction steps during Montgomery Multiplication are separated or integrated,
- form of the multiplication and reduction steps.

In this section we will highlight two of these algorithms: Separated Operand Scanning (SOS) and Coarsely Integrated Operand Scanning (CIOS). While all algorithms in [19] require same number of word-level multiplications, the number of additions, memory read and write operations differ in each. The CIOS method is the most efficient and fastest method when implemented on general purpose microprocessors, since it has the least amount of memory space with $s+3$ words, where s is the number of words in one operand, and requires less addition, read and write operations. However, a modified version of SOS method is implemented for *cryptographically-enhanced processor* and the reasons for choosing the SOS method is analyzed in Section 3.1.4.

3.1.2 The Separated Operand Scanning (SOS) Method

The SOS Method (cf. Algorithm 3) consists of two separate steps: multiplication and reduction. First multiplication of two integers is performed and then product is reduced to its final form. Because the outer loop moves through words of one of the operands during the execution of algorithm, the method is called as operand scanning [19].

The first part of the algorithm is a school-book multiplication which computes $2s$ word size product and stores in t where s is the number of words in the operands. Then value of u is then computed as follows according to the second step in Algorithm 2

$$u = (t + m \cdot n)/r$$

where $m = t \cdot n' \mod r$. First u is taken as $u = t$, then $m \cdot n$ is added to u and finally u is divided by r, which is simply shifting u to right or ignoring lower words of u [19]. The ADD function in the method performs the carry propagation operation. Since carry can propagate to the last word, one bit carry may be generated at the end and this carry should be stored. Storing the final carry increases the size of t by one word and size of t becomes $2s+1$ words. Finally value of u is stored in $s+1$ words and if the value of u is greater than the modulus, u is subtracted from modulus and final value of multiplication is found.

The analysis in [19] demonstrates that during the SOS method, following number of operations have to be performed

- $2s^2 + 2$ multiplications
- $4s^2 + 4s + 2$ additions
- $6s^2 + 7s + 3$ reads
- $2s^2 + 6s + 2$ writes

Furthermore, it is noted that the SOS Method requires a total of $2s+2$ words for temporary results. $2s + 1$ of these words is used for storing t and one word is used for storing the value of m.

Algorithm 3 Separated Operand Scanning (SOS) Method

Input: a, b, n multi-word integers (w bits in each word),
s: number of words in the operands and modulus
Output: t: multi-word product

1. **for** $i = 0$ to $s - 1$
2. $C = 0$
3. **for** $j = 0$ to $s - 1$
4. $(C, S) = t[i + j] + a[j] \cdot b[i] + C$
5. $t[i + j] = S$
6. $t[i + s] = C$
7. **for** $i = 0$ to $s - 1$
8. $C = 0$
9. $m = t[i] \cdot n'[0] \mod 2^w$
10. **for** $j = 0$ to $s - 1$
11. $(C, S) = t[i + j] + m \cdot n[j] + C$
12. $t[i + j] = S$
13. $\mathbf{ADD}(t[i + s], C)$
14. **for** $j = 0$ to s
15. $u[j] = t[j + s]$

3.1.3 The Coarsely Integrated Operand Scanning (CIOS) Method

CIOS method (cf. Algorithm 4) differs from the SOS method in a way that the CIOS integrates both multiplication and reduction steps. Instead of computing entire multiplication, the CIOS method alternates during the iterations of the outer loops of multiplication and reduction. Integration of multiplication and reduction is possible since the value of m during the ith iteration of the outer loop for reduction depends only on the value of $t[i]$ and this value is already computed by i^{th} iteration of the outer loop for multiplication [19].

The analysis in [19] reveals the number operations executed while performing modular multiplication with the CIOS method are as follows

- $2s^2 + s$ multiplications
- $4s^2 + 4s + 2$ additions
- $6s^2 + 7s + 2$ reads
- $2s^2 + 5s + 1$ writes

Moreover, it is shown in [19] that the CIOS method reduces memory usage significantly when compared to the SOS method. The SOS method uses $2s+2$ words for storage of temporary results while the CIOS method requires only $s + 3$ words where $s + 2$ words are used to store t and one word is used for storing m.

Algorithm 4 Coarsely Integrated Operand Scanning (CIOS) method

Input: a, b, n multi-word integers (w bits in each word),
s: number of words in the operands and modulus
Output: t: multi-word product

1. **for** $i = 0$ to $s - 1$
2. $\quad C = 0$
3. $\quad$ **for** $j = 0$ to $s - 1$
4. $\quad\quad (C, S) = t[i + j] + a[j] \cdot b[i] + C$
5. $\quad\quad t[j] = S$
6. $\quad (C, S) = t[s] + C$
7. $\quad t[s] = S$
8. $\quad t[s + 1] = C$
9. $\quad C = 0$
10. $\quad m = t[i] \cdot n'[0] \mod 2^w$
11. $\quad$ **for** $j = 0$ to $s - 1$
12. $\quad\quad (C, S) = t[i + j] + m \cdot n[j] + C$
13. $\quad\quad t[j] = S$
14. $\quad (C, S) = t[s] + C$
15. $\quad t[s] = S$
16. $\quad t[s + 1] = t[s + 1] + C$
17. $\quad$ **for** $j = 0$ to s
18. $\quad\quad t[j] = t[j + 1]$

3.1.4 Enhanced SOS Method

Fastest and most efficient Montgomery multiplication on a general-purpose processor can be implemented by using the CIOS method according to the analysis provided in [19]. However, in *cryptographically-enhanced processor,* cryptographic operations are executed in the proposed *CEU* which is different than execution units of general purpose computers. The *CEU* requires that all operands should be stored in the *CRF*, therefore a new analysis should be performed for the CIOS and SOS methods

The SOS method separates the multiplication and reduction operations and first performs the multiplication and then reduction. For the worst case, which is performing 1024-bit modular multiplication in RSA, all operands and the product can fit into the *CRF* after the multiplication step since the *CRF* has a total size of 512 Bytes. For the reduction step modulus and m value should be stored in the *CRF* and these values can be written over the operands since after multiplication step, operands are not used anymore (cf. Algorithm 3).

However, the CIOS method integrates both multiplication and reduction step and executes them interleaved. Therefore all operands, product, modulus and m as well should be stored during the execution of the CIOS method (cf. Algorithm 4). Storing all these values require 544 Bytes of space which is larger than the size of *CRF*. For this reason, a modified version of SOS method is implemented for performing modular multiplications on *cryptographically-enhanced processor.*

The Enhanced SOS Method is presented in Algorithm 5. In the enhanced method all multiplications are computed as 128-bit multiplications and the

product is stored in HI and LO registers. Therefore for the addition operations HI and LO registers are used.

Algorithm 5 Enhanced SOS Method

Input: a, b, n multi-word integers (w bits in each word),
s: number of words in the operands and modulus
Output: t: multi-word product

1. **for** $i = 0$ to $s - 1$
2. $\quad C = 0$
3. $\quad$ **for** $j = 0$ to $s - 1$
4. $\quad\quad (C, S) = t[i + j] + LO + C \quad (a[j] \cdot b[i] \rightarrow HI \,||\, LO)$
5. $\quad\quad t[i + j] = LO$
6. $\quad\quad C = HI + Carry$
7. $\quad t[i + s] = C$
8. **for** $i = 0$ to $s - 1$
9. $\quad C = 0$
10. $\quad m = t[i] \cdot n^{'}[0] \mod 2^w$
11. $\quad$ **for** $j = 0$ to $s - 1$
12. $\quad\quad (C, S) = t[i + j] + LO + C \quad (m \cdot n[j] \rightarrow HI \,||\, LO)$
13. $\quad\quad t[i + j] = LO$
14. $\quad$ **ADD**$(t[i + s], C)$
15. **for** $j = 0$ to s
16. $\quad u[j] = t[j + s]$

3.1.5 Performance Analysis

In this section performance analysis of the CIOS and the SOS method is provided. Modular Multiplication is heavily performed both in RSA and elliptic curve cryptography, therefore operand sizes are chosen according to the security levels of both algorithms. Typical unbreakable and secure RSA key length is 1024-bits and the same level of security for the elliptic curve cryptography can be achieved by using 160-bit key length . Therefore, in the performance analysis operand sizes are chosen starting from 160-bit and up to 1024-bits.

Performance of the CIOS method is tested on *base processor* since in [19] it is suggested that the CIOS method is the most efficient method for hardware implementation and processors. The performance of the SOS method is tested on *cryptographically-enhanced processor* to utilize the proposed enhancements. Performance of both algorithms in clock cycles and speedup values for modular multiplication is presented in Table 5 and 6. Table 5 provides the speedup values for 5 stage pipeline versions of *base processor* and *cryptographically-enhanced processor* and Table 6 provides speedup values for 7 stage pipeline versions of both processors.

Note that execution time of 160 and 192-bit multiplications in the SOS method is greater than the execution time of 256-bits. The reason for this deviation is due to the width of registers in the *CRF*. Since the *CRF* has 128-bit width, after multiplication step in the SOS, 160 and 192-bit multiplication results have to aligned before reduction which increases the execution time.

Precision	CIOS	SOS	Speedup
160	2,765	1,047	2.6
192	3,873	1,196	3.2
256	6,691	931	7.2
512	25,605	2,365	10.8
1024	100,304	7,654	13.1

Table 5: Speedups for Modular Multiplication on 5-stage pipeline version

Precision	CIOS	SOS	Speedup
160	3,032	1,132	2.7
192	3,747	1,282	2.9
256	7,310	1,013	7.2
512	27,856	2,598	10.7
1024	108,493	8,418	12.9

Table 6: Speedups for Modular Multiplication on 7-stage pipeline version

4 Modular Inversion

4.1 Modular Inversion in finite $GF(p)$

Basic modular arithmetic operations such as addition, multiplication and inversion have significant importance on numerous cryptographic systems. The RSA algorithm [25], Diffie-Hellman key exchange algorithm [8], US Government Digital Signal Standard [23] and Elliptic Curve Cryptography [18, 20] are some examples in which modular arithmetic operations are heavily performed.

Modular inversion operation is crucial in public key cryptography since it is used for accelerating the so-called addition-subtraction chains [10, 16] and computing point operations on an elliptic curve defined over finite field $GF(p)$[18, 20] .

In elliptic curve cryptography when affine coordinates are used, inversion in $GF(p)$ becomes the most time consuming operation. Projective coordinates can be used yet one inversion is still necessary for conversion of the result to a desired representation. Even a single inversion operation brings a significant overhead on a general-purpose processor.

Modular inverse of an integer $a \epsilon [1, p-1]$ in $GF(p)$ is denoted as the integer $x \epsilon [1, p-1]$ which has the following property:

$$a \cdot x = 1 \mod p$$

The most common and one of the best ways to compute modular inverse in finite fields is the binary extended Euclidean Algorithm[16]. However, for the hardware implementation, most efficient inversion algorithms are Kaliski

[15] and its variation Montgomery Inversion algorithms [26].

4.1.1 Kaliski and Montgomery Inversion Algorithm

The modular inverse of an integer, a, is redefined by Kaliski in his work [15]. Kaliski inversion algorithm computes modular inverse of an integer by using the principles of the Montgomery arithmetic introduced by [22]. The Kaliski inverse of the integer (cf. Algorithm 6) is defined in *residue* domain and compatible with the Montgomery arithmetic and given as follows:

$$Kaliski_{Inv}(a) = a^{-1}2^n \mod p$$

where p is a prime number and n is the number of bits in prime p ($i.e.\, n = \lceil log_2 p \rceil$) .

Kaliski inversion algorithm consists of two phases. Phase I calculates the integer r such that

$$r = a^{-1}2^k \mod p$$

where $n \leq k \leq 2n$. Phase II is merely a correction step and computes the integer x which is the inverse of a in the *residue* domain

$$x = a^{-1}2^n \mod p$$

The number of iterations in Phase II is entirely dependent on k, which is found to be about *1.4*n where n is the length of the modulus in bits. The k value is appropriate according to the analysis performed in [16] for the number of halving in the binary extended Euclidean algorithm.

Montgomery Inversion algorithm [26] is similar to the Kaliski Inversion algorithm. Phase I steps are exactly the same for two algorithms. The difference between two algorithms is the output value at the end of Phase II. Modular inverse of the integer a in the Kaliski inversion is computed as

$$x = Kaliski_{Inv}(a) = a^{-1}2^{n} \mod p$$

while in the Montgomery inversion the inverse value of a is computed as

$$x = Montgomery_{Inv}(a) = a^{-1}2^{2n} \mod p$$

The Kaliski algorithm is suitable for calculating inverse of a number represented in integer domain since inverse of an integer a is computed as

$$x = Kaliski_{Inv}(a) = a^{-1}2^{n} \mod p$$

However, if the input of the Kaliski inversion algorithm is given in the *residue* domain, $a2^n$, then inverse of the number in residue form is computed as

$$x = Kaliski_{Inv}(a2^{n}) = a^{-1} \mod p$$

which is in the integer domain.

In order to have fast elliptic curve operations, operations are performed using the Montgomery arithmetic thus numbers should be represented in the *residue* domain. If one uses the Kaliski inversion for a number represented in the *residue* domain, some extra arithmetic operations are needed to convert result from integer to the *residue* domain. Instead, the Montgomery inversion

(cf. Algorithm 7) can be used for the numbers represented in *residue* domain. In Montgomery Inversion, the inverse of a number in *residue* domain, $a2^n$, is represented as

$$x = Montgomery_{Inv}(a2^n) = a^{-1}2^n \mod p$$

which is also in the *residue* domain.

Both algorithms can be used interchangeably to compute inverse in *residue* domain. Kaliski Inversion can be used to compute inverse of a number represented in integer domain, while Montgomery Inversion can be employed for computing inverse of numbers in *residue* domain.

Algorithm 6 Kaliski Inversion Algorithm

Phase I

Input: $a \epsilon [p-1]$ and prime p

Output: $r \epsilon [p-1]$ and k, where $r = a^{-1}2^k \mod p$ and $n \leq k \leq 2n$

1. $u = p, v = a, r = 0$,$s = 1$ and $k = 1$
2. **while** $(v > 0)$
3. **if** u is even **then** $u = u/2$**,** $s = 2s$
4. **else if** v is even **then** $v = v/2$**,** $r = r/2$
5. **else if** $u > v$ **then** $u = (u - v)/2$**,** $r = r + s$**,** $s = 2s$
6. **else** $v = (v - u)/2$**,** $s = s + r$**,** $r = 2r$
7. $k = k + 1$
8. **if** $r \geq p$ **then** $r = r - p \mod p$
9. **return** $r = p - r$ and k

Phase II

Input : $r \epsilon [1, p-1]$**,** p and k

Output : $x \epsilon [1, p-1]$**,** where $x = a^{-1}2^n \mod p$

1. **for** $i = 0$ **to** $k - n$
2. **if** r is even **then** $r = r/2$
3. **else** $r = (r + p)/2$
4. **return** $x = r$

Algorithm 7 Montgomery Inversion Algorithm

Phase I

Input: $a \,\epsilon\, [p-1]$ and prime p

Output: $r \,\epsilon\, [p-1]$ and k, where $r = a^{-1}2^k \mod p$ and $n \leq k \leq 2n$

1. $u = p, v = a, r = 0$,$s = 1$ and $k = 1$
2. **while** $(v > 0)$
3. **if** u is even **then** $u = u/2$**,** $s = 2s$
4. **else if** v is even **then** $v = v/2$**,** $r = r/2$
5. **else if** $u > v$ **then** $u = (u - v)/2$**,** $r = r + s$**,** $s = 2s$
6. **else** $v = (v - u)/2$**,** $s = s + r$**,** $r = 2r$
7. $k = k + 1$
8. **if** $r \geq p$ **then** $r = r - p \mod p$
9. **return** $r = p - r$ and k

Phase II

Input : $r \,\epsilon\, [1, p-1]$**,** p and k

Output : $x \,\epsilon\, [1, p-1]$**,** where $x = a^{-1}2^{2n} \mod p$

1. **for** $i = 0$ **to** $2n - k$
2. $r = 2r$
3. **if** $(r \geq p)$ **then** $r = r - p$
4. **return** $x = r$

4.1.2 Implementation Details

The most time consuming operations performed during Kaliski and Montgomery inversion are multi-precision addition,subtraction, division and multiplication by 2 operations, which can also be observed in Algorithm 3 and 4. The *IU* inside the *CEU* is designed to increase the performance of these operations. Addition and subtraction operation is performed by 128-bit adder circuit. Multiplication and division by 2 are implemented as shift left and shift right operations on microprocessors. However, shift operations in the *cryptographically-enhanced processor* slightly differs from the traditional processors. Shift operations are performed on the values loaded from the *CRF* and the *CRF* has two read ports but only one write port. Hence shift operations are performed as shifting boundary bits (i.e. least or most significant bits) from one register to another while only one of the registers changes (cf. Table 2).

4.1.3 Performance Analysis

In this section execution times of Kaliski and Montgomery Inversion algorithms on *base processor* and *cryptographically-enhanced processor* are provided in terms of clock cycles. Cycle times are taken from both 5 and 7 stage pipeline versions of the processors and presented in the tables below. The first values in the tables represent the execution time on 5 stage pipeline version of given processor while second values denote the execution time on 7 stage pipeline version. Tables 7 and 9 demonstrate the execution time of Montgomery and Kaliski Inversion operation on the *base processor* respectively. Execution time for the same algorithms on the *cryptographically-enhanced*

processor are given in Tables 8 and 10. Finally speed-up values for both Montgomery and Kaliski Inversion operations are given in Tables 11 and 12.

Precision	Phase I	Phase II	Total
160	63,855 / 77,168	14,319 / 16,848	78,174 / 94,016
192	86,300 / 104,400	19,782 / 22,896	106,082 / 127,296
256	140,624 / 170,424	31,783 / 37,515	172,407 / 207,939
512	468,936 / 566,448	110,942 / 131,656	579,878 / 698,104

Table 7: Montgomery Inversion on *base processor*

Precision	Phase I	Phase II	Total
160	29,153 / 32,012	7,825 / 8,777	36,978 / 40,789
192	34,706 / 38,185	9,158 / 10,300	43,864 / 48,485
256	45,552 / 50,259	11,616 / 13,095	57,168 / 63,354
512	114,447 / 123,942	27,389 / 29,879	141,836 / 153,821

Table 8: Montgomery Inversion on *cryptographically-enhanced processor*

Precision	Phase I	Phase II	Total
160	63,906 / 77,041	8,504 / 9,832	72,410 / 86,873
192	86,313 / 104,135	11,432 / 13,369	97,745 / 117,504
256	141,075 / 170,311	18,573 / 22,100	159,648 / 192,411
512	473,306 / 570,196	59,594 / 70,340	532,900 / 640,536

Table 9: Kaliski Inversion on *base processor*

Precision	Phase I	Phase II	Total cycles
160	29,560 / 32,318	3,374 / 3,679	32,934 / 35,997
192	35,103 / 38,463	3,894 / 4,281	38,997 / 42,744
256	45,961 / 50,483	4,779 / 5,089	50,740 / 55,572
512	116,814 / 126,613	11,198 / 11,939	128,012 / 138,552

Table 10: Kaliski Inversion on *cryptographically-enhanced processor*

Precision	Phase I	Phase II	Total
160	2.19 / 2.41	1.83 / 1.92	2.11 / 2.30
192	2.49 / 2.73	2.16 / 2.22	2.42 / 2.63
256	3.09 / 3.39	2.74 / 2.86	3.02 / 3.28
512	4.10 / 4.57	4.05 / 4.41	4.09 / 4.54

Table 11: Montgomery Inversion Speedups

Precision	Phase I	Phase II	Total
160	2.16 / 2.38	2.52 / 2.67	2.20 / 2.41
192	2.46 / 2.71	2.94 / 3.12	2.51 / 2.75
256	3.07 / 3.37	3.89 / 4.34	3.15 / 3.46
512	4.05 / 4.50	5.32 / 5.89	4.16 / 4.62

Table 12: Kaliski Inversion Speedups

5 Implementation Details

In this section, we provide the speedup values obtained for both RSA and elliptic curve cryptography by using the proposed enhancements in Chapters 3 and 4. Also we will demonstrate *time* $\times$ *area* metric for RSA and elliptic curve cryptography the *cryptographically-enhanced processor* synthesized on Avnet LX200 FPGA board.

We implement a simple 1024-bit RSA using two methods: windowing method with 4-bit window size and no windowing. On the *base processor*, 1024-bit RSA with 4-bit windows takes on average $132,361,636$ clock cycles. The profile of the operation reveals that 97.48% of execution time is spent on modular multiplication. Consequently, the speedup obtained by running same operation on the *cryptographically-enhanced processor* is 11.26. 1024-bit RSA with no windowing method takes $156,812,860$ clock cycles on *base processor*, 97.86% of which is spent on modular multiplication according to the profile information. The speedup for this case is found out to be 11.47.

Similarly, we implemented elliptic curve scalar point multiplication with jacobian coordinates [6] and the implementation results are given in Table 13.

Precision	*base processor*	*enhanced processor*	% of Multiplication	Speedup
160	5,684,844	2,695,097	87	2.11
192	9,774,069	3,673,000	90.17	2.66
256	21,509,576	4,412,633	92.49	4.87
512	160,109,439	19,798,812	96.51	8.08

Table 13: Implementation Results for Elliptic Curve Point Multiplication

It is a common tendency to think that there is no need to speedup inversion operation due to projective coordinates (e.g. jacobian coordinates) [6]. Projective coordinates eliminate all but one inversion from elliptic curve point operations at the expense of more multiplications; only inversion operation is needed for converting from projective coordinates to affine coordinates. We demonstrate, in this section, that the time spent on one inversion might be significant especially when the modular multiplication is performed on our enhanced processor. Using projective coordinates, one elliptic curve scalar point multiplication takes approximately $2,695,097$ clock cycles for 160-bit elliptic curve on the enhanced processor. Our implementation of Montgomery inversion for 160-bit operands would consume, on the other hand, $76,692$ clock cycles if the ISA is not utilized. The inversion, therefore, consumes only about 2.8% of all clock cycles spent on scalar point multiplication including conversion. This does not call for a strong need to speedup the inversion operation since any improvement on inversion will marginally speedup the entire point operation. There are, however, pre-computation techniques that significantly improve the elliptic curve point operations. For example, with fixed-base comb method it is possible to perform one scalar point multiplication in expectedly $342,901$ clock cycles on the enhanced processor. This time, the inversion operation would consume about 22.36% of clock cycles without the CU; which is a good motivation for speeding up inversion operation. Consequently, this would be translated into 11.78% speedup in one point multiplication due to the improvement in inversion calculations.

5.1 FPGA Emulation and Time-Area Metrics

Tensilica supports implementation of LX2 cores on both ASIC's and FPGA boards. Since ASIC implementation of the *cryptographically-enhanced processor* is costly and time consuming process, the *cryptographically-enhanced processor* is synthesized on FPGA boards. Tensilica supports only Avnet LX60 and Avnet LX200 boards for implementation of LX2 processor cores. Among two boards, Avnet LX200 board is chosen for the realization of *cryptographically-enhanced processor* due to its high flexibility and greater number of slices.

Implementation of the *cryptographically-enhanced processor* on Avnet LX200 board is as follows: first configuration of the *cryptographically-enhanced processor* with TIE files attached to processor, which defines the architecture extensions, are sent to Tensilica Xtensa Processor Generator (XPG) server. XPG generates and supplies precompiled bitstreams for Avnet LX200 boards. Prior to sending the *cryptographically-enhanced processor* to XPG, target frequency is set to 50 MHz, which is the largest target frequency for Avnet LX200 boards. Both the *base processor* and the *cryptographically-enhanced processor* is sent to XPG. According to the bitstreams, the frequency of the *base processor* is 50.6 MHz while frequency of the *cryptographically-enhanced processor* is 50.3 MHz. Furthermore, the *base processor* occupies $18,224$ slices on Avnet LX200 boards and the *cryptographically-enhanced processor* occupies $31,530$ slices on the same FPGA board. These results show that architectural enhancements increase the core size without affecting the critical path delay.

By utilizing the information obtained from the bitstreams of both proces-

sors, $time \times area$ metric for RSA and elliptic curve scalar point multiplication is presented in tables below. Table 14 presents the $time \times area$ product of RSA for both no windowing and 4-bit windowing options. The product values are normalized to the 4-bit windowing implementation of RSA on the *cryptographically-enhanced processor.*

Core	Operation	Area (Slices)	Clock Cycles	Time × Area
Base	RSA (4-bit w.)	18,224	132,361,636	**6.50**
Enhanced	RSA (4-bit w.)	31,530	11,753,299	**1.0**
Base	RSA (no w.)	18,224	156,812,860	**7.71**
Enhanced	RSA (no w.)	31,530	13,642,547	**1.16**

Table 14: $Time \times Area$ product for RSA

Table 15 provides the $time \times area$ product values of point multiplication operations for elliptic curve cryptography (ECC). Note that, the product values are normalized to the 160-bit point multiplication operation on *cryptographically-enhanced processor.*

Core	Operation	Area (Slices)	Clock Cycles	Time × Area
Base	160-bit ECC	18,224	5,684,844	**1.22**
Base	192-bit ECC	18,224	9,774,069	**2.10**
Base	256-bit ECC	18,224	21,509,576	**4.61**
Base	512-bit ECC	18,224	160,109,439	**34.34**
Enhanced	160-bit ECC	31,530	2,695,097	**1.00**
Enhanced	192-bit ECC	31,530	3,673,000	**1.36**
Enhanced	256-bit ECC	31,530	4,412,633	**1.64**
Enhanced	512-bit ECC	31,530	19,748,196	**7.33**

Table 15: $Time \times Area$ for ECC

Finally, Table 16 presents the performance improvement of RSA and ECC point multiplication operation.

Operation	Improvement
RSA (4-bit w.)	**6.51**
RSA (no w.)	**6.64**
160-bit ECC	**1.22**
192-bit ECC	**1.54**
256-bit ECC	**2.82**
512-bit ECC	**4.69**

Table 16: Improvements for RSA and ECC

6 An AES Implementation Hardened Against Cache Attacks

Efficient software implementations of many symmetric key ciphers are vulnerable to cache attacks since they usually utilize look up tables for nonlinear function (s-box) calculations, where these look up tables generally fit in the first or second-level caches of the modern processors. The most efficient AES implementation in software is due to Barreto [4] where four 1 KB tables are used for the first nine round of 128-bit AES. Another table of the same size (can actually be made smaller, i.e. 256 B) is used in the last round. Many cache-based attacks [1, 2, 7] exploit access patterns of cryptographic process to cache lines, which may contain the desired data value (cache hit), or not (cache miss). Considering the fact that a cache miss introduces a significant (and observable) delay to the computation due to the fact that cache memory is much faster than the main memory, the discrepancies in the execution time of different runs of AES leak information on the secret (or round) keys. For a formal model of cache attacks, one can profitably refer to [5, 24].

Most powerful attacks focus either on the first round of AES as in [24] and on the last round as in [1] since these rounds directly interact with outside world by taking the plaintext and outputting the ciphertext, both of which are easily observable by an adversary. Therefore, it is of utmost importance to protect the first and last rounds. Implementing even one round without using any look up tables (in order not to leave any trace in the cache) can be painfully slow in software due to involved bit manipulation operations.

A fast AES implementation secure against cache attacks may have to use a

combination of the techniques proposed in [5, 24]. Our proposed architecture can be beneficial in applying these protection mechanisms. For example, the aforementioned *CRF* can be used to store a part of the s-box since its limited size does not lend itself to accommodate all look up tables, which are 5 KB in total size. 256 bytes of the *CRF* can only hold up precomputed values for byte substitution layer of one AES round. Since lookups in this table result in accesses to the register file, which returns the requested byte in constant time, there will be no timing differences. Furthermore, the fact that the *CRF* is not time shared (at least while cryptographic process has locked it), other (possibly spy) processes cannot observe the trace left by the cryptographic process.

A small look up table, specifically placed for AES implementation and organized for a fast access enables that any round of the AES can be securely implemented without any significant overhead in timing when compared to the implementation in [4]. However, our design principle is to recycle versatile cryptographic units to benefit various cryptographic operations. Since the timing requirements for public key algorithms is more demanding, we elect to organize the *CRF* as one-dimensional array of 32 elements of 128 bit each. The drawback of this approach is some overhead in accessing the desired bytes in the register file. One can, all the same, always arrange the *CRF* as byte array if symmetric cipher performance is more important.

We modified the implementation in [4] by replacing the rounds with their secure counterparts, and listed the overhead (in number of clock cycles) roundwise in Table 17 for a single block encryption of 128 bits.

[4]	1st	last	$1st$ + last	per round
796	171(21.5%)	33(4.5%)	199(25%)	178(≈ 22.4%)

Table 17: Overhead of protecting rounds of AES in number of clock cycles

To give an idea as to how expensive is to protect even one round of AES, we also implemented an AES version without any look up table in software optimized for speed; the resulting overhead of protecting only the first round of AES turns out to be 36, 125 clock cycles.

As explained in [5] for standard implementation, a small table can be used to protect the rounds of AES. We also implemented the standard method for different rounds and found out that protecting one round brings about the overhead of approximately 29%, which is higher than our . Consequently, the standard implementation whose all rounds are protected takes about 2654 clock cycles, while our fully-protected implementation on the enhanced core takes about 2, 246. This translates into 16% improvement over the standard implementation. Note that the standard implementation may be still vulnerable to *synchronized attacks* through a spy process that can evict cache lines during the AES computation. On the other hand, our AES implementation provides perfect security against cache attacks and costs almost no overhead in hardware since we utilize the cryptographic unit which is already included for public key operations.

The case study of AES confirms our claim as to the necessity of a cryptographic register file that can be used for multiple purposes, e.g. speeding up the cryptographic operations, storing sensitive key elements, and secure implementation of non-linear functions (s-boxes).

7 Conclusion and Future Work

We designed and implemented a cryptographic unit (CU), for secure and fast execution of a wide range of cryptographic algorithms. The proposed CU introduces new functional units and instructions for performing faster multi-precision arithmetic operations on finite fields. To show the design's efficiency and applicability, we integrated the proposed CU into the execution pipeline of a low cost, configurable and extensible embedded processor core. We obtained considerable speedups for basic multi-precision arithmetic operations such as modular multiplication and inversion in $GF(p)$ finite field, which are the dominant operations in many public key cryptosystems. Impact of the speedups are demonstrated for scalar point multiplication operation of elliptic curve cryptography and RSA. Up to 8 and 10 times of performance improvements are achieved respectively for these public key cryptosystems.

We synthesized the extended processor core on a specific FPGA board to observe hardware cost of the CU. Synthesis results reveal a conservative hardware cost for the CU. However, additional hardware does not result a significant increase in critical path delay. $Time \times area$ product of RSA and elliptic curve cryptography shows a performance enhancement up to 6 and 4 times respectively. A comparison of the obtained speedup values and incurred hardware overhead clearly confirms that the benefits of the CU far exceed its cost. Furthermore, we showed that CU can be used to harden software implementations of symmetric-key ciphers with low overhead against certain side-channel attacks (i.e. cache attacks).

We leave the actual implementation of the enhanced embedded processor on ASIC's as a future work. Higher frequency values can be achieved

on ASIC implementations. Throughout the design process of *CU,* all the enhancements introduced are designed in such a way that they do not have significantly adverse effect on the critical path of the base processor. For instance, the critical path of a 128-bit multiplier consists of four 32-bit multipliers that work in parallel to each other. We applied the same approach for the other functional units in the *CU*. There will be, however, definitely an associated penalty in the maximum applicable frequency in ASIC implementation due to the increase in the total chip area. Limited reduction in clock frequency would not be a major problem for embedded applications where relatively low clock frequencies are adopted. Exploring the (possibly negative) effect of the *CU* on the maximum applicable frequency and optimizing the functional units to minimize this effect is left as a future work.

References

[1] Onur Aciiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES (short paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer Verlag, Berlin, Germany, 2006.

[2] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache-Based Remote Timing Attacks on the AES. In MAsayuki Abe, editor, *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer Verlag, Berlin, Germany, 2007.

[3] ARC. Arc configurable cores. Website. `http://www.arc.com/configurablecores`.

[4] P. Barreto. The AES Block Cipher in C++. Website, 2003. `http://planeta.terra.com.br/informatica/`.

[5] Johannes Blömer and Volker Krummel. Analysis of Countermeasures Against Access Driven Cache Attacks on AES. In *Selected Areas in Cryptography*, pages 96–109, 2007.

[6] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *ASIACRYPT '98: Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security*, pages 51–65, London, UK, 1998. Springer-Verlag.

[7] D. Bernstein. Cache-Timing Attacks on AES. Website, 2005. http://cr.yp.to/papers.html#cachetiming.

[8] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. IEEE *Transactions on Information Theory*, IT–22(6):644—654, 1976.

[9] Nikil Dutt and Kiyoung Choi. Configurable processors for embedded computing. *Computer*, 36(1):120–123, 2003.

[10] Omer Egecioglu and Cetin Kaya Koc. Exponentiation using canonical recoding. *Theoretical Computer Science*, 129(2):407–417, 1994.

[11] A. Murat Fiskiran and Ruby B. Lee. On-Chip Lookup Tables for Fast Symmetric-Key Encryption. In *ASAP*, pages 356–363, 2005.

[12] Johann Großschädl and Erkay Savas. Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m). In *CHES*, pages 133–147, 2004.

[13] Johann Großschädl, Stefan Tillich, and Alexander Szekely. Performance Evaluation of Instruction Set Extensions for Long Integer Modular Arithmetic on a SPARC V8 Processor. In *DSD*, pages 680–689, 2007.

[14] Improv. Improv configurable dsp. Website. http://www.improvsys.com.

[15] Burton S. Kaliski. The montgomery inverse and its applications. *IEEE Trans. Comput.*, 44(8):1064–1065, 1995.

[16] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1981.

[17] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[18] N. Koblitz. Elliptic curve cryptosystems. *Math. Computing*, 48:203–209, 1987.

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

[20] Alfred J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Boston:Kluwer Academic Publishers, 1993.

[21] V.S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology — Crypto '85*, pages 417–426, New York, 1986. Springer-Verlag.

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

[23] National Institute of Standards and Technology. Digital signature standard (dss). Federal information processing standards publication 186-2, 2000.

[24] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, pages 1–20, 2006.

[25] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[26] E. Savas and Ç. K. Koç. The montgomery modular inverse-revisited. *IEEE Trans. Comput.*, 49(7):763–766, 2000.

[27] Tensilica. Tensilica:configurable and standard processor cores for soc design. Website. `http://www.tensilica.com/`.

[28] Tensilica. The xtensa lx2 architecture. Website. `http://www.tensilica.com/products/lx_architecture.htm`.

[29] Tensilica. Xtensa lx2 embedded processor core. Website. `http://www.tensilica.com/products/xtensa_LX.htm`.

[30] Stefan Tillich and Johann Großschädl. Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In *CHES*, pages 270–284, 2006.

[31] Tobias Vejda, Dan Page, and Johann Großschädl. Instruction Set Extensions for Pairing-Based Cryptography. In *Pairing*, pages 208–224, 2007.