COMPACT, FLEXIBLE AND FAST COPROCESSOR DESIGN FOR ELLIPTIC CURVE PAIRING OPERATION ON RECONFIGURABLE HARDWARE

by

ERTUĞRUL MURAT

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 2011

COMPACT, FLEXIBLE AND FAST COPROCESSOR DESIGN FOR ELLIPTIC CURVE PAIRING OPERATION ON RECONFIGURABLE HARDWARE

APPROVED BY:

Associate Prof. Dr. Erkay Savaş: .....

(Thesis Advisor)

Associate Prof. Dr. Albert Levi: .....

Associate Prof. Dr. Cem Güneri: .....

Associate Prof. Dr. Yücel Saygın: .....

Assistant Prof. Dr. Selim Balcısoy: .....

DATE OF APPROVAL: …….

ABSTRACT

Proposal of Identity-Based cryptography by Shamir in 1984 opened a new area for researchers. Failing to provide a feasible implementation of identity based encryption (IBE), Shamir developed a signature scheme, whereby signatures can be verified by publicly available information such as signer's identity. Since the first efficient implementation of IBE realized using pairing operation on elliptic curves due to Boneh and Franklin a plethora of papers has been published and many studies have been conducted covering different aspects of pairing-based cryptography. Today, pairing is used in many cryptographic applications including, identity based cryptography, key exchange protocols, short signatures, anonymous signatures and in many other newly emerging protocols and schemes. Also, pairing is still a developing research field yielding important challenges for the research community.

Pairing computation involves fairly complicated operations compared to classical symmetric and asymmetric cryptosystems. Multitudes of pairing types have been proposed after its first appearance in the literature. Also, each of them involves selection of many parameters such as the choice of the underlying field and its characteristics, order of the embedding degree, type of the elliptic curve etc. Therefore, different types of optimisations are possible rendering selection process extremely difficult. Because of the abundance of choices, for an efficient pairing implementation many criteria have to be examined. For instance, selection of pairing type, construction of finite fields and elliptic curves, coordinate systems to represent points on the curve and algorithms and architecture for arithmetic operations play a crucial role on the performance of the specific implementation of the pairing-based cryptography.

A multitude of implementations regarding to pairing-based cryptography have been proposed in the literature. However, most of them are software realizations; the reason being is the complexity of the overall system. Some hardware implementations have already been proposed, but most of them are very specific, therefore lacks flexibility and scalability. Due to the complexity of the system, some researches advice to use dedicated implementations for specific set of parameters even in software, limiting the flexibility of the implementation further.

In this thesis, we propose a very generic, flexible and compact hardware co-processor for all kinds of pairing implementations intended for implementation on reconfigurable devices (e.g. FPGA). Our co-processor supports all types of pairing operations with different parameter classes via making use of highly-optimized hardware implementations of basic arithmetic operations common not only to pairing operations, but also to elliptic curve cryptography and other public key cryptography algorithms. Our design utilizes the idea of hardware-software co-design concept. To accelerate pairing computation we implement some units responsible for performing the most time-consuming operations as a generic, but highly optimized hardware circuits, whereas we prefer to implement some complex parts (unworthy of hardware resources) in low-level software of micro-instructions. Although we use two arithmetic cores running concurrently, our design still manages to be compact thanks to its careful and generic design.

ÖZET

Kimlik-temelli kriptografik sistemin 1984'te Shamir tarafından ortaya atılmasıyla, araştırmacılar için yeni bir kapı aralanmış oldu. Kimlik-temelli şifreleme işlemi için uygulanabilir bir algoritma önermeyen Shamir, imzanın geçerliliğinin imzalayanın herkese açık bilgileriyle, örneğin kimliği, doğrulanabildiği uygulanabilir bir elektronik imzalama sistemi geliştirdi. Kimlik-temelli şifrelemenin ilk uygulanabilir örneğinin Boneh ve Frankin tarafından eliptik eğriler üzerinde tanımlanmış eşleme (pairing) işlemi ile verilmesinden bu yana, kriptografi alanında eşleme temelli pek çok çalışmalar yapılıp, yayınlar çıktı. Günümüzde eşleme operasyonu pek çok kriptografik uygulamada kullanılmaktadır, kimlik temelli kriptografik sistemler, anahtar değişim protokolleri, kısa imzalar, anonim imzalar ve yeni gelişen pek çok protokol ve uygulama bunların arasındadır. Özet olarak kriptografik eşleme, içerisinde çözülmesi gereken birçok problemi barındıran ve halen gelişen bir araştırma alanıdır.

Eşleme operasyonu klasik simetrik ve asimetrik kriptografik sistemlere göre oldukça karmaşıktır. İlk eşleme operasyonunun geliştirilmesinden bu yana eşleme operasyonunun birçok sayıda türevi çıkmıştır. Her bir türev kullanılan cebrik cismin seçimi ve onun karakteristiği, yerleştirme derecesi gibi birçok parametre kullanmaktadır. Bundan dolayı parametre seçim sürecini oldukça zorlaştıran bir hayli optimizasyon bulunmaktadır. Seçenek bolluğundan dolayı etkili bir eşleme operasyonu gerçeklemesi için pek çok ölçüt incelenmelidir. Örneğin, eşleme işleminin tipi, uygun cebrik cismin ve eliptik eğrinin seçimi, kullanılacak koordinat sisteminin, algoritmaların ve aritmetik operasyonlar için donanım mimarilerinin seçimi gibi konular eşleme operasyonunun etkin gerçeklenmesinde önemli rol oynamaktadır.

Literatürde pek çok eşleme işlemi gerçeklemesi mevcuttur; fakat bunların çoğu salt yazılımsal gerçeklemelerdir. Bunun sebebi gerçeklenen operasyonun karmaşıklığıdır. Bunlar dışında bazı donanımsal gerçeklemeler mevcutsa da bunların çoğu çok özelleşmiş uygulamalardır ve bu nedenle esneklik ve ölçeklenirlikten yoksundur. Operasyonun karmaşıklığından dolayı bazı araştırmacılar verimli bir gerçeklemeye sahip olmak için yazılımsal dahi olsa, tasarımın esnekliğini sınırlayarak, özelleşmiş tasarımlara gidilmesini salık vermektedir.

Bu tezde, programlanabilir donanım cihazlarında gerçekleştirilmek üzere, her türde eşleme operasyonları için çok esnek, genel ve kompakt bir yardımcı-işlemci tasarımı sunulmaktadır. Geliştirilen tasarım, değişik parametre sınıflarında her eşleme operasyonu türevini desteklemektedir. Bunu yaparken sadece eşleme operasyonu için değil, diğer birçok asimetrik anahtarlı şifreleme sistemlerinde de kullanılan temel aritmetik operasyonları gerçekleyen son derece optimize edilmiş donanımsal işlevsel birimler kullanmaktadır. Tasarımda ortaya koyduğumuz yaklaşım, yazılım ve donanımın ortak kullanımıdır. Eşleme operasyonunu hızlandırmak için en çok zaman harcayan operasyonlar parametrik ve oldukça optimize donanımsal birimler olarak gerçeklenirken, karmaşık operasyonlar (kısıtlı donanım kaynaklarını verimli olarak kullanamayan) mikro-operasyonlar vasıtasıyla yazılımsal olarak gerçeklenmiştir. Tasarımda her ne kadar eş zamanlı çalışan ve aritmetik işlemleri gerçekleyen iki-çekirdek kullanılsa da, dikkatli tasarım ve esnek yapı sayesinde tasarım karşılaştırmalı olarak az yer kaplamaktadır.

Dedicated to my family…

ACKNOWLEDGEMENTS

I would like to present my special thanks to my thesis advisor, Associate Prof. Dr. Erkay Savaş for his valuable mentorship, not only about this thesis but also for his guidance in general manner. He helped me in all points that I cannot make progress. For all the difficult corners of this thesis he became very elucidative. I also thank to members of my thesis jury, Associate Prof. Dr. Albert Levi, Associate Prof. Dr. Cem Güneri, Associate Prof. Dr. Yücel Saygın and Assistant Prof. Dr. Selim Balcısoy, for very useful suggestions on my thesis. Besides I would like to thank to Ersin Öksüzoğlu for sharing his valuable work, Mongomery multiplier, with me. I also sincerely thank to Ali Can Atıcı for all his helps during design process.

Last but not least, I thank to my family for their unlimited support. They are the ones who helped me stay where I stay in all respects. I do not forget the friends whom I did not count the names but who are always with me and fortify me. I thank to all.

Table of Contents

List of Terms and Symbols

- **ACIU:** Arithmetic core and inversion unit.
- **ASIC:** Application Specific Integrated Circuit
- **BMC:** Block of micro code.
- **BRAM** : Block RAM; hardwired RAM in FPGA.
- **CIOS:** Coarsely Integrated Operand Scanning
- **DLP:** Discrete logarithm problem.
- **DMA:** Direct memory access
- **DSP48A1:** Hardwired arithmetic unit in FPGA
- **DSS:** Digital Signature Standard
- **FDEU:** Fetch decode and execute unit.
- **FPGA:** Field Programmable Gate Array
- **LSW:** Least significant word. If a variable is thought as sequence of words having same bit size each, then LSW defines the least significant word.
- **LUT:** Both stands for number of LUTs and look up tables: Boolean function generators in FPGA
- **M:** Modulus
- **m:** bit size of modulus.
- **ms:** milliseconds: 10^{-3} seconds.
- **MF:** Maximum frequency; achievable maximum frequency in an FPGA design.
- **MM:** Montgomery multiplier: A special multiplier specialized for hardware.
- **MSW:** Most significant word.

- **Opcode:** Operation code. This is the part of the micro code which defines what kind of operation to be executed.
- **PAR:** Place and route: Last step in the implementation before embedding the core.
- **REG:** Flip flop numbers used in a design.
- **T:** Total time to complete the operation
- **TA:** Time are product; LUT*T/1000
- **us:** microseconds: 10^{-6} seconds.
- **WL:** Word length; bit size of a processing word.

List of Figures

List of Tables

List of Algorithms

1 Introduction

Most commonly accepted definition of the pairing operation is as follows: Pairing is a bilinear map which is defined from $\boldsymbol{G_1} \times \boldsymbol{G_2}$ to $\boldsymbol{G_T}$, $(\boldsymbol{G_1} \times \boldsymbol{G_2} \rightarrow \boldsymbol{G_T})$, where $\boldsymbol{G_1}$ and $\boldsymbol{G_2}$ are usually additive groups implemented on elliptic curves and $\boldsymbol{G_T}$ is multiplicative group [3]. Pairing is first introduced to cryptographic community by Menezes et al., with a destructive example, MOV attack [1]. In their study, they propose a method for converting discrete logarithm problem, which is defined over an elliptic curve on a finite field F_p, to the discrete logarithm problem over an extension field $F_{p^k}^*$. However, real take off in pairing is realized with application of pairing to the identity-based cryptography (IBC) by Boneh and Franklin [2]. Since then, pairing has been a very active research topic with multitude of papers published every year. Pairing is mainly used in IBC, certificate-less cryptosystem, in key agreement protocols [10], [11] and many new cryptographic applications [12].

Many pairing types are proposed in the literature [13], [14], [15]. Also many optimization methods are proposed for operations in pairings to efficiently implement it in hardware and software [16], [17], [13]. However, most studies are about software implementations of pairings [18], [19], [20]. There are some publications which aim hardware realizations, but they are few in number and besides, it is very difficult to find common points among them to make a fair comparison. This is due to the fact that, each implementation uses a special type of pairing or special parameters. There is a multitude of parameters that affect the efficiency and scalability of a pairing implementation; both in hardware or software. Some of the parameters includes: type of the curve, type of the coordinate systems used for elliptic curve point representation, underlying field, and extension degree of the fields, and even hamming weight of an input variable [3].

In this thesis, we design a general-purpose pairing coprocessor for arbitrary elliptic curves and embedding degrees targeted for reconfigurable hardware implementation. We propose a balanced mixture of hardware-software methods and architectures for realization of pairing operation. It aims to use advantages of both software and hardware. While hardware is very efficient in realizing some dedicated operations that constitute the computational bottleneck of the pairing operation (e.g. field multiplication), it is a valuable resource and cannot be easily spent on complex operations, which are not worthy of hardware resources. At this point software remedies the situation by providing cost-effective solutions to complex operations, even though it is not as fast as hardware. We aim to propose an architecture that can fit into small and old fashioned FPGAs, like Xilinx Spartan 3S400 [21]; and when used with very modest middle range FPGAs, like Xilinx Spartan-6SLX45T [5], there remains plenty of implementation space for other purposes. However, being small is not the only goal of the design; an acceptable speed performance is required. Our processor employs two arithmetic cores, which provide shorter operation time by using parallelization. In addition to these, our design is parametric and very flexible. It provides trade-off between area and speed in a very wide spectrum. According to design privileges, design can be easily changed from an area-efficient design to speed-efficient design. Variables that facilitate the flexibility of our design are listed below:

- Word Length (WL): Our processor operates over variables of words similar to a general-purpose CPU. However, our word size is changeable. This parameter defines the bit length of the word.
- Input Length (IL): Some dedicated hardware implementations are designed to operate on a constant input size. However, our design can easily be adapted to work on different input lengths. This parameter defines the total bit length of the longest input variable (e.g modulus in modular arithmtetic).
- Pipeline Stage Number (PSN): This parameter defines the total number of pipeline stages used in multiplier for the underlying prime field, which is an important part of the design.

Main subject of this study is a pairing processor, as previously mentioned, since many parameters affect the efficiency of pairing operation. We also need suitable parameters and curves to work on.

Pairing operation can be realized over certain classes of elliptic curves satisfying some special parameters, as explained in [4], and detailed in the next chapter, are known as pairing-friendly elliptic curves. Pairing operation involves arithmetic over an extension field, thus we have to decide and find a suitable elliptic curve and extension field to use in our implementations.

In addition, we also have to be careful about the efficiency and security of the system. One parameter that directly affects the security and efficiency of the system is the bit length of prime integer (*the* modulus) for the field over which we construct our elliptic curve. As bit length of the modulus increases, arithmetic operations begin to slow down, but security increases. Another factor that affects the security and speed is the embedding degree of elliptic curve, which is also the degree of irreducible polynomial that the extension field is built upon. As embedding degree gets bigger that can increase the security level, complexity of arithmetic operations in the extension field increases.

One of the optimizations to reduce the execution time of pairing is proposed for extension field multiplication. We use Karatsuba-Ofman [22] algorithm to reduce multiplication time in the extension field. Before completing pairing operation, an exponentiation operation has to be done on extension field. Here again we use an optimized method to considerably decrease the total exponentiation time.

Pairing is an operation defined over elliptic curves whereby choice of the coordinate systems is important for efficiency reasons. For example, in affine coordinate system during elliptic curve point addition and point doubling, a division operation has to be performed. But the division is very time consuming operation. Therefore, we have to choose a coordinate system that does not need division. We prefer to use Jacobian mixed projective coordinate system as it needs no division operation during point addition and point doubling. Moreover, it exhibits better performance than other projective coordinate systems.

In the next section we provide the details about the underlying FPGA architecture, selection of elliptic curves, extension field operations and elliptic curve arithmetic operations. Also Tate pairing is explained in detail and some optimization techniques are discussed to reduce the overall running time of the algorithm.

2 Underlying FPGA Architecture & Background Information

In this section we provide information about the structure of the FPGA which we use to implement our co-processor architecture. Also we give information about pairing operation in general and Tate pairing in particular. We choose to implement our design in Spartan-6SLX45T, due to the fact that it is a low-cost middle-range FPGA, meaning it does not have abundance of logic resources like high-end FPGA devices, but has a modest level of logic resources close to low-end FPGA devices [5]. Another reason is that Xilinx Spartan-6 family members are optimized for low power consumption. In the following subsection underlying FPGA architecture is discussed.

2.1 Underlying FPGA Architecture

Spartan-6 provides low power solutions with its 45 nm manufacturing technology. It provides low power consumption with high performance with the help of its 1.2 V core voltage. Compared to the previous members of Spartan family, its power consumption is as low as half of theirs. Also, it provides moderate logic resources [5]. One member of the Spartan-6 family, Spartan-6SLX45T, is 84.4$ today, whereas a cheap and older FPGA, Spartan-3S400 costs about 31$ [7]. However, Spartan-6 has five to six times more logic resources than Spartan-3. Therefore, cost of per logic unit in Spartan-6 is lower than the cheapest FPGA. Hence Spartan-6 offers the best price-performance ratio compared to the older Spartan family. If we look at all the advantages, Spartan-6 appears as a good choice for low-cost, low-power embedded cryptographic applications, which necessitate considerably complicated operations.

Understanding architecture and capabilities of underlying FPGA architecture is essential for efficient designs. This is only possible provided that complete insight of FPGA attributes is available to make right decisions about the design.

There are several special building blocks inside the Xilinx Spartan-6 FPGA, which we use in our design. These are configurable logic blocks (CLBs), block RAMs (BRAM) and digital signal processor units (DSP48A1s). These components provide flexibility in the design and efficient use of resources.

CLBs are the main reconfigurable logic block of the FPGA. One CLB contains two slices and every slice contains four look-up tables (LUTs) and eight flip-flops. LUTs are mostly known as Boolean function generators of the FPGA. However, they can also serve as RAM and shift register. LUTs in Spartan-6 have six inputs and two output ports. These LUTs are, in fact, composed of two smaller, five-input LUTs. Therefore, with one LUT either two five-input logic functions or a six-input logic function can be realized. There are several types of slices; SLICEX, SLICEL and SLICEM. Differences between them are as follows: SLICEX is the simplest one, where LUTs are only capable of realizing logic functions. It does not contain arithmetic structure, nor can it be used as shifter or RAM. SLICEL contains carry-logic and its LUTs can be combined to construct large multiplexers. SLICEM is the most functional one. In addition to the functions in SLICEL, LUTs in SLICEM can be used as distributed RAM and shifter. Both SLICEL and SLICEM feature carry look-ahead logic for fast addition operation. By default, addition is implemented using carry look-ahead adder logic in the FPGA. Thus, we do not use any structure other than the one automatically inferred by the FPGA for addition. Trying to implement addition by using other logic resources does not result in a better adder due to the fact that default adder type of FPGA is already carry look-ahead adder, moreover it is placed into a specialized area. What is meant by specialized area is that logic elements used in carry generation have very low latency values.

Since there are four LUTs in a slice, a 4-bit adder/subtractor is easily realized within a slice. For operands larger than four bits, a special structure reduces the latency in carry generation path. Normally, slices in a CLB are not directly connected to each other; they are connected to a switching matrix outside the FPGA, as can be seen in

Figure 1. After switching matrix, they connect to global routing resources; then appropriate routing is achieved.

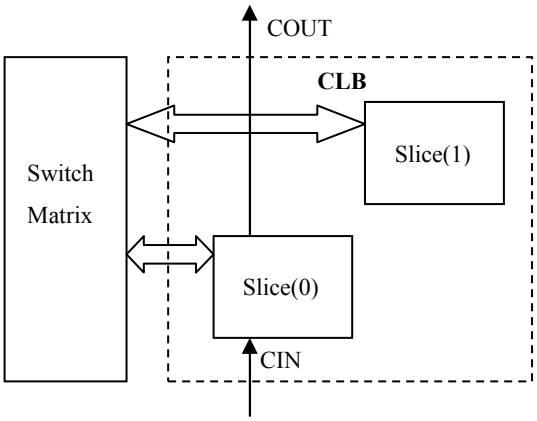

Figure 1: Connection of Slices [6]

However in the case of carry propagation, carry output of one slice directly connects to the carry input of the other slice. Hence, fast propagation of carry is possible. This situation is depicted in Figure 2.

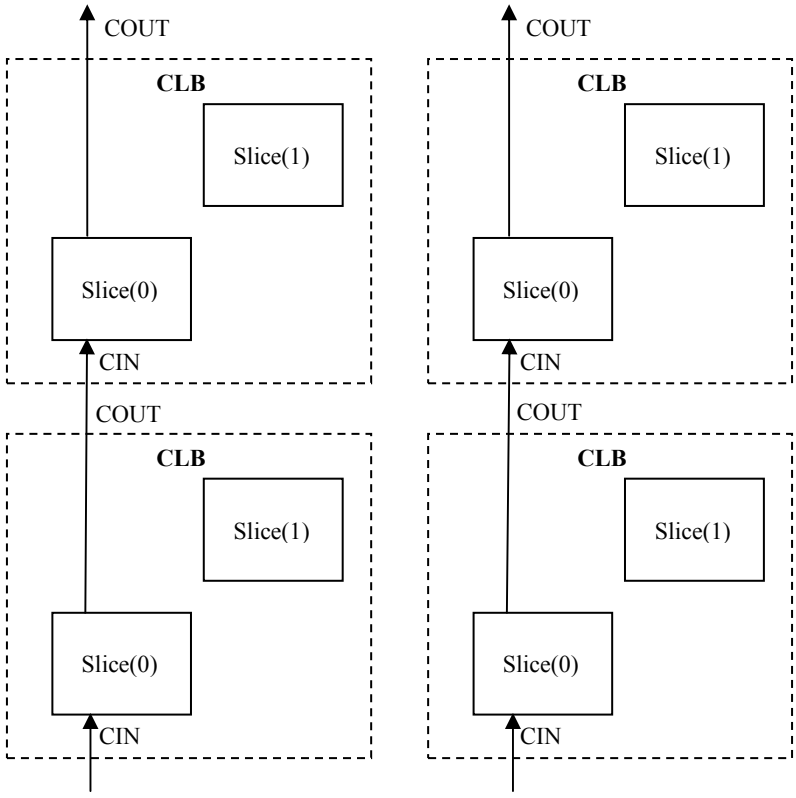

Figure 2: Inter CLB Carry Propagation [6]

The feature related to carry propagation is not new to Spartan-6, while it exists even in older Spartan-3 family; the implementation is much faster in Spartan-6.

LUTs have many other useful features. LUTs can be configured to construct wide multiplexers. As previously mentioned, LUTs in Spartan-6 have six inputs which enable us to realize a (4×1) multiplexer in one LUT. Thus, when using multiplexers equal or smaller than (4×1), only one LUT is used. It is important to keep this property in mind and trying not to use larger multiplexers than (4×1). For example, when a (5×1) multiplexer is used logic usage doubles rather than a linear increase. To implement multiplexers having sizes between (5×1) to (8×1), we need the same amount of LUTs, which is double of (4×1) in this case. We understand that this is important especially when we think about multiplexers used in large data buses. Number of multiplexer utilized for one bit switching is multiplied with size of bus in a multiplexer used in bus switching.

Another important feature of LUTs is that they can be configured as *distributed RAM*. However, only LUTs in SLICEMs can be used as RAM. These LUTs have some additional attributes that enable them to act like a RAM. They have inputs for data as well as a write enable. Thus in most basic version, they can be configured as single port, 64×1 RAM with synchronous write and asynchronous read. Nevertheless, their output can be made synchronous by using the flip-flops in SLICEM. RAM, that is constructed using LUTs are called distributed RAM. Distributed RAM and BRAM can be employed interchangeably [6].

BRAMs are hardwired memory blocks inside the FPGA. They have synchronous read/write operations. A BRAM can have different widths and depths. Wider BRAMs are automatically formed by the implementation tool. BRAMs are utilized generally when a need for high memory usage arises. Especially when big variables are used, like in our case and generally in most cryptographic applications, employing of BRAMs saves significant amount of logic resources. BRAMs have fixed places in the FPGA which is actually physically in the middle of FPGA. This may cause some unexpected latency in some cases when circuit is placed away from the BRAMs. In these cases outputs and inputs of the BRAMs should be registered [8].

DSP48A1 is a special hardwired block for arithmetic and logic operations. There are equivalent functional units in older versions of FPGAs. It contains hardwired and

pipelined adders/subtractors and multipliers. In our design we use 18×18 hardwired multipliers. We do not use hardwired adders/subtractors inside of DSP48A1, since CLBs also have specialized carry logic as explained previously. Moreover using DSP48A1 for addition/subtraction may cause some extra delay due to routing to resources. To overcome this problem registered inputs and outputs are usually used. In this case registers adds extra clock cycles at each access of source and this increases the overall processing time. This is not worthwhile in case of adder/subtractor. On the other hand, since implementing multiplier with logic resources consumes too much area, we use DSP48A1 units for performing multiplication [9].

2.2 Background Information on Algebraic Structures

G_1 and G_2 are two additive groups and G_3 is a multiplicative group. And let all of them have a group order *r*, which can be further assumed to be prime number. Then pairing is a map defined as follows: $e: G_1 \times G_2 \rightarrow G_3$, which satisfies the following properties, given that P is a generator of G_1 and Q is a point on G_2, which is linearly independent of P [23]:

1. **Bilinearity**: For all $P, R \in G_1$ and for all $Q, S \in G_2$

$$e(P+R, Q) = e(P, Q) \times e(R, Q)$$
$$e(P, Q+S) = e(P, Q) \times e(P, S)$$
$$e(iP, Q) = e(P, Q)^i \; and \; e(P, iQ) = e(P, Q)^i$$

   where $\times$ denotes the multiplication in G_3.

2. **Non-degeneracy:** For all $P \in G_1 - \{0\}$, there exists some $Q \in G_2$ such that; $e(P, Q) \neq 1$ and for all $Q \in G_2 - \{0\}$, there exists some $P \in G_1$ such that; $e(P, Q) \neq 1$

Tate pairing over elliptic curves is one type of the pairing operation that can be calculated efficiently and satisfies the aforementioned properties. P and Q are chosen as follows: Let F_q is a prime field and $E(F_q)$ is curve over that field. Let r be a prime such that, there exists a point on the elliptic curve $E(F_q)$ with order of *r*. Moreover $r \mid \#(E(F_q))$ [24] where $\#(E(F_q))$ denotes the number points on the elliptic curve. Let k be the smallest number satisfying $r \mid q^k - 1$, and $r \nmid q^l - 1$ for $1 \leq l < k$

[25]. The integer k is referred as the embedding degree of $E(F_q)$. Set of the points on $E(F_q)$ of order r is denoted as $E(F_q)[r]$. Then $P \in E(F_q)[r]$ and $Q \in E(F_{q^k})$ are the inputs of the Tate pairing operation. More precisely, Tate pairing is defined as a map $e: E(F_q)[r] \times E(F_{q^k}) \to F_{q^k}^*/(F_{q^k}^*)^r$ and considered as the evaluation of a rational function f_P, whose divisor is $div(f_P) = r[P] - r[\infty]$ ([∞] is point at infinity), such that:

$$e(P, Q) = f_P(D_Q)^{q^k - 1/r},$$

where $D_Q \sim [Q] - [\infty]$ is the divisor for Q [24] (for more information about divisors see [26]).

The most efficient implementations for pairing computation use Miller's algorithm proposed in [27], which evaluates the rational function f_P at point Q. Tate pairing algorithm consists of elliptic curve and polynomial arithmetic operations over finite fields. Without any optimizations, the computation becomes prohibitively time-consuming. One of the algorithms that computes Tate pairing efficiently is BKLS algorithm [28], as described in Algorithm 1.

Algorithm 1: BKLS Tate Pairing Algorithm [4]

Inputs: $P, Q \in E$ and $r \in \boldsymbol{Z}$
$T \leftarrow P, f \leftarrow 1$
Output: $f_{r,P}(Q)^{\frac{p^k-1}{r}}$
1. for $i = \lfloor \lg(r) \rfloor - 2$ to 0
2. $f \leftarrow f^2 * l_{T,T}(Q)$
3. $T \leftarrow [2]T$
4. if $r_i = 1$ then
5. $f \leftarrow f * l_{T,P}(Q)$
6. $T \leftarrow P + T$
7. end if
8. end for
9. $f \leftarrow f^{\frac{p^k-1}{r}}$

Many possible optimizations exist for Algorithm 1. Some optimizations are possible for arithmetic operations over F_{q^k}, for evaluation of line computation function $l_{A,B}(Y)$ (steps 2 and 5), for elliptic curve operations (point addition and point

doubling) (steps 3 and 6) and for final exponentiation operation (step 9). Moreover even selection of proper r value can be included into these optimizations.

Potential optimizations are explained in the next subsection. But prior to this finding the appropriate elliptic curve and pairing parameters are detailed since Tate pairing performance also depends on these parameters.

2.2.1 Finding Tate Pairing Parameters

We choose to operate on a field with embedding degree being $k = 4$. Although another embedding degree can be selected for different security requirements we believe that this degree provides optimum security-complexity trade-off. Security of a pairing operation depends on two parameters: The bit size of the subgroup in elliptic curve, which is $log_2 r$, and the bit size of extension field, which is $k * log_2 q$. Values of these parameters should be chosen according to the best known attack towards them. Most successful attack for elliptic curve discrete logarithm problem (ECDLP) is Pollard-ρ technique whose complexity is $\mathcal{O}(\sqrt{r})$ [29]. On the other hand best attack to prime extension fields, F_{q^k}, is index-calculus method whose complexity is given by; $\mathcal{O}(L_{q^k}(1/3))$ and $L_{q^k}(1/3) = exp((32/9)^{1/3} * (log\ q^k)^{1/3} * (loglog\ q^k)^{2/3})$ [31]. According to NIST suggestions [30] for 80 bit security it is proper to choose r as a 160-bit integer and q^k as 1024-bit integer. We choose r as 160 bits and q as 256 bits for 80 bits security following the NIST's advice. However choosing the bit length is only one aspect of the task, since all together r, q and k should satisfy some equations explained as in section 2.2. We use the following formulas proposed in [32] to find appropriate r, q values for $k = 4$.

$$t(x) = -4x^3$$

$$r(x) = 4x^4 + 4x^3 + 2x^2 + 2x + 1$$

$$q(x) = \frac{1}{3}(16x^6 + 8x^4 + 4x^3 + 4x^2 + 4x + 1)$$

Some other formulas can be used but above equations give the whole set of elliptic curves whose embedding degree $k = 4$ and having a discriminant value equals

to 3 (as explained in subsequent sections). With the help of a software program using these equations for desired bit lengths, r and q values can be found. Note that both and r and q are prime numbers, so for each value found, primality test have to be run.

Another point to note is that extension fields are built using irreducible polynomials whereas q is just the prime of the field so we have to choose an irreducible polynomial. Since degree of our extension field is $k = 4$ then the degree of the irreducible polynomial should be 4. We choose a small irreducible polynomial in the form of $x^k - \beta$ to simplify the extension field arithmetic operations. In our case β is 2 since it is a small number and moreover multiplying a number with 2 means shifting it to the left by 1 bit, which is a very easy operation compared to multiplication. Thus, another constraint is added to check when pricking a suitable q: To make $x^k - 2$ irreducible polynomial, 2 should be quadratic non-residue in modulo q. In the equations, $t(x)$ represents the trace of elliptic curve. As can be remembered r should divide $\#E(F_q)$, which is equal to $q + 1 - t$. This variable is used in finding elliptic curve in next section.

2.2.2 Finding Elliptic Curve

After finding q, r and t values we can build an ordinary elliptic curve using these parameters. We use following elliptic curve equation: $y^2 \equiv x^3 + a * x + b \ (mod\ q)$, where $a = a_0 * k_a$ and $b = b_0 * k_b$. To find elliptic curve variables a, b, IEEE 1363 standard [33], which defines standards for elliptic curve cryptography, is used. According to the standard for a given discriminant a_0 and b_0 values are predetermined and k_a and k_b values are random. Since we already choose our discriminant value as 3, a_0 and b_0 values are known. Table 1 shows the values of a_0 and b_0 for given discriminants:

D	a_0	b_0
1	1	0
2	-30	56
3	0	1
7	-35	98
11	-264	1694

Table 1: a_0 and b_0 Values for Discriminant [33]

We define another variable, k', for finding proper elliptic curve. This value comes from Hasse's theorem; $k' * r = \#E(F_q) = q + 1 - t$ since we know the right hand side of the equation we can compute k' and curve parameters can be calculated using q, k', r and a_0, b_0. Note that k' has no relation with embedding degree k. Curve parameters and generator point of $\#E(F_q)$, P, can be found using Algorithm 2 defined in IEEE 1363.

Algorithm 2: Finding the Curve and Generator Point [33]

Inputs: EC parameters p, r and k' and coefficients a_0, b_0

Output: A curve E modulo q and a generator point P on E with order r, or a "wrong order" message

1. Select an integer δ s.t. $0 < \delta < q$
2. If D = 1, then $a \leftarrow a_0 \delta^4 \bmod q$ and $b \leftarrow 0$; if D=3 then $a \leftarrow 0$ and $b \leftarrow b_0 \delta^6 \bmod q$. Otherwise, $a \leftarrow a_0 \delta^2 \bmod q$ and $b \leftarrow b_0 \delta^3 \bmod q$
3. Look for a point P of order r on the curve $y^2 = x^3 + a * x + b \ (mod\ q)$ via A.11.3
4. If the output of A.11.3 is "wrong order", then output the message "wrong order" and stop
5. Output the coefficient a, b and the point P.

Selection of δ in the first step of the algorithm relies on the kind of coefficients wanted. For instance:

- If D≠1 or 3, and it is wanted $a = -3$, then δ is taken as the solution to $a_0 \delta^2 \equiv -3 \ (mod\ q)$ if there exists. If does not exists or selection of δ causes a message "wrong order", then choose another curve as follows. If $q \equiv 3 \ (mod\ 4)$ and the result was "wrong order" then choose $-\delta \bmod q$ instead of δ; the result leads to a curve with $a = -3$ and the right order. If no solution δ exists, or if $q \equiv 1 \ (mod\ 4)$, then repeat A.14.4.1 with another root of the reduced class polynomial. The ratio of roots leading to a curve with $a = -3$ and the right order is roughly one-half if $q \equiv 3 \ (mod\ 4)$, and one-quarter if $q \equiv 1 \ (mod\ 4)$.
- If there is no restriction on coefficients, then choose δ at random. If it turns out "wrong order", then repeat the algorithm till a set of parameters a, b and P is obtained. This occurs for half the values of δ, unless D=1 (one quarter of values) or D=3 (one-sixth of values)

For Step 3 of Algorithm 2, where a base point is found, is given in Algorithm 3.

Algorithm 3: Finding a Point P of Order r [33]
Inputs: A prime r, a positive integer k' not divisible by r, an elliptic curve $E(F_q)$ **Output:** If $\#(E(F_q)) = k'r$, a P on E with order r, If not, "wrong order" message 1. Generate a random point P (not $[\infty]$) on E 2. $P \leftarrow k'P$ 3. If $P = [\infty]$ then go to step 1 4. $P' \leftarrow rP$ 5. If $P' \neq [\infty]$ then output "wrong order" and stop 6. Return P

Using the Algorithms 2 and 3 an elliptic curve $E(F_q)$ and a generator point P is found. As can be remembered there is no constraint on the point Q, other than being linearly independent of P. Thus, it is easy to find a Q point for starting the Tate pairing operation [23].

2.2.3 Polynomial Arithmetic for F_{q^k}

The values f and $l_{A,B}(Q)$ in Algorithm 1 are in F_{q^k}. Thus there are considerably high numbers of polynomial operations for arithmetic of F_{q^k}. Most time-consuming operation of them is the inversion; but due to the algorithm used [4], denominator elimination can be applied. At the end of the Miller's loop (**for** loop) in the Algorithm 1, denominator of the variable$f$ goes to 1. Thus, there is no need to perform inversion during the Miller's loop. Therefore multiplication stands as the most time consuming operation in the Miller's loop. We use an optimized method, called Karatsuba multiplication method [22], to reduce the number of F_q multiplications used to perform F_{q^k} multiplications. The method is summarized as follows [34]. Let A and B be polynomials of degree $k-1$, with k coefficients:

$$A(x) = \sum_{i=0}^{k-1} a_i x^i \, , B(x) = \sum_{i=0}^{k-1} b_i x^i$$

For each $i = 0,1, \ldots\ldots, k-1$ the terms $D_i \coloneqq a_i b_i$ are computed. Also, for $j = 1, 2, \ldots, 2k-3$ and for all s and t given $s + t = j$ and $t > s \geq 0$ the following terms are calculated

$$D_{s,t} \coloneqq (a_s + a_t)(b_s + b_t)$$

Afterwards, $C(x) = A(x)B(x) = \sum_{i=0}^{2k-2} c_i x^i$ can be calculated as follows:

$$c_0 = D_0 \, , \, c_{2k-2} = D_{k-1}$$

$$c_i(x) = \begin{cases} \sum_{s+t=i;t>s\geq 0} D_{s,t} - \sum_{s+t=i;t>s\geq 0}(D_s + D_t) \, , for\ odd\ i; 0 < i < 2k-2 \\ \sum_{s+t=i;t>s\geq 0} D_{s,t} - \sum_{s+t=i;t>s\geq 0}(D_s + D_t) + D_{i/2}, for\ even\ i; 0 < i < 2k-2 \end{cases}$$

Rightness of the formula and its complexity are discussed in [35]. This method requires $\mathcal{O}(1/2(k^2 + k))$ multiplications inF_q while classical method requires $\mathcal{O}(k^2)$ to perform one F_{q^k} multiplication.

In calculation of F_{q^4} multiplication we use two Karatsuba multiplications recursively. First we calculate F_{q^2} multiplication using F_q multiplication, for which explicit formulas used, when $k = 2$, is given in Algorithm 4. We build F_{q^2} over F_q as $F_{q^2} = F_q[x]/(x^2 - \beta)$, where β is a quadratic non-residue in F_q.

Algorithm 4: Implementation of Karatsuba method on F_{q^2}

Inputs: $a = a_0 + a_1 i, b = b_0 + b_1 i$
Output: $c = a * b$ where $c = c_0 + c_1 i$.

1. $t_1 = a_0 b_0$
2. $t_2 = a_1 b_1$
3. $t_3 = \beta t_2$
4. $c_0 = t_1 + t_3 = a_0 b_0 + \beta a_1 b_1$
5. $t_3 = t_1 + t_2 = a_0 b_0 + a_1 b_1$
6. $t_1 = a_1 + a_0$
7. $t_2 = b_1 + b_0$
8. $t_4 = t_1 t_2 = (a_0 + a_1)(b_0 + b_1)$
9. $c_1 = t_4 - t_3 = (a_0 + a_1)(b_0 + b_1) - (a_0 b_0 + a_1 b_1)$

- Total cost of the operation: $4F_q$ multiplication + $5F_q$ addition

Then we implement F_{q^4} multiplication using F_{q^2} multiplications. F_{q^4} field is built upon F_{q^2} field using tower construction. $F_{q^4} = F_{q^2}[y]/(y^2 - \gamma)$ and $F_{q^2} =$

$F_q[x]/(x^2-\beta)$ where $i=\sqrt{\beta}\in F_{q^2}$ and $\gamma=i$ where $I=\sqrt{i}=\sqrt{\gamma}\in F_{q^4}$. This type of construction is called tower field. The tower field construction makes things easier in extension field operations. Thus, we can effectively build F_{q^4} operations over F_{q^2} operations. The method for F_{q^4} multiplication is given in Algorithm 5.

Algorithm 5: Implementation of Karatsuba method on F_{q^4}
Inputs: $A=A_0+A_1I$, B$=B_0+B_1I;\ A_0,A_1,B_0,B_1\ \in F_{q^2}$ **Output:** $C=A*B$ where $C=C_0+C_1I;\ C_0,C_1\ \in F_{q^2}$ 1. $T_1=A_0B_0$ 2. $T_2=A_1B_1$ 3. $T_3=\gamma T_2=i(t_{2,0}+t_{2,1}i)=\beta t_{2,1}+t_{2,0}i$ 4. $C_0=T_1+T_3=A_0B_0+\gamma A_1B_1$ 5. $T_3=T_1+T_2=A_0B_0+A_1B_1$ 6. $T_1=A_1+A_0$ 7. $T_2=B_1+B_0$ 8. $T_4=T_1T_2$ 9. $C_1=T_4-T_3$ - Total cost of the operation: $3F_{q^2}$ multiplication + $1F_q$ multiplication + $5F_{q^2}$ addition

2.2.4 Elliptic Curve Arithmetic on Projective Coordinates

We use Jacobian mixed coordinate system since in Algorithm 1, point P is in affine coordinate system. This coordinate system is more effective than other projective coordinate systems in terms of overall (both doubling and addition) operation count [45]. Another reason for using projective coordinate systems is to eliminate division (inversion), which is the most time consuming operation, in affine coordinate systems. A point $T=(x_1,y_1,z_1)$ in projective coordinate system corresponds to the point $P=(x_1/{z_1}^2,y_1/{z_1}^3)$ in affine coordinate system. Point doubling formulas for point $A=(x_A,y_A,z_A)$ for the curve $y^2=x^3+ax+b$ is given as follows. $C=2A=(x_c,y_c,z_c)$ then $\lambda_{2A}=3{x_A}^2+a{z_A}^4$ where λ_{2A} is slope of tangent.

$$x_c={\lambda_{2A}}^2-8x_A{y_A}^2$$

$$y_c=\lambda_{2A}(4x_A{y_A}^2-x_C)-8{y_A}^4$$

$$z_C=2y_Az_A$$

Addition formula for points $A = (x_A, y_A, z_A)$ and $B = (x_B, y_B, 1)$ where $C = A + B = (x_c, y_c, z_c)$ and $\lambda_{A,B} = (y_A - {z_A}^3 y_B)$ is the slope of line AB.

$$z_C = ({z_A}^2 x_B - x_A) z_A$$

$$x_C = (y_A - {z_A}^3 y_B)^2 - ({z_A}^2 x_B + x_A)(x_A - {z_A}^2 x_B)^2$$

$$y_C = {z_C}^2 \lambda_{A,B} ({z_C}^3 x_B - z_c x_c) - {z_C}^3 y_B$$

Please note that denominators of the results are not given because, denominator of f goes to 1 at the end of the algorithm thanks to denominator elimination property. So we never compute denominators.

2.2.5 Line Evaluation Function

The function denoted by $l_{A,B}(Q)$ in Algorithm 1 is known as *line evaluation function*. Geometrically it is the distance between the point Q and the line that intersects the points A and B [36]. Formulas related to $l_{A,B}(Q)$ and $l_{A,A}(Q)$ are given as follows:

$$l_{A,B}(Q) = \left(\mathrm{y_Q z_A}^3 - \mathrm{y_A}\right)\mathrm{z_C} - \lambda_{A,B}(\mathrm{x_Q z_A}^3 - \mathrm{x_A z_A})$$

Formula for $l_{A,A}(Q)$ is the same as above except that λ_{2A} is used instead of $\lambda_{A,B}$. As might be remembered $Q = (x_Q, y_Q)$ is in $E(F_{q^k})$, and therefore, line computation involves arithmetic in $E(F_{q^k})$, which is costly. However, there is a trick to make the computation of line evaluation much easier. Instead of using the full point Q on $E(F_{q^k})$, we can use the twist of $E(F_{q^k})$ in a smaller field such as $E'(F_{q^{k/d}})$, where d is a proper integer that divides k. The elliptic curve $E'(F_{q^{k/d}})$ can be called as the twist of $E(F_{q^k})$ if there exists an isomorphism between them such that ψ: $E'(F_{q^{k/d}}) \rightarrow E(F_{q^k})$ [37]. Since our embedding degree is 4 we can choose d as 2 and in this case twist is named as *quadratic twist*, which is defined as follows:

$$E'(F_{q^2}): y^2 = x^3 + aw^{-2}x + bw^{-3}, a, b \in F_q;\ w \in F_{q^2}$$

where w is a quadratic non-residue in F_{q^2} thus $\sqrt{w} \in F_{q^4}$ and the isomorphism is given by [38]:

$$\psi_2 : \begin{cases} E'(F_{q^2}) \rightarrow E(F_{q^4}) \\ (x, y) \rightarrow (xw, yw^{3/2}) \end{cases}$$

Thus by using the twist curve, we can choose coordinates of the point Q on F_{q^2} instead of choosing them on F_{q^4}. The twisted coordinates $y_Q^{'}, x_Q^{'} \in F_{q^2}$ are the coordinates on twist such that $y_Q^{'} = (0 + 0i) + (y_{Q_{10}}^{'} + y_{Q\_11}^{'} i)I$ and $x_Q^{'} = \left(x_{Q_{00}}^{'} + x_{Q_{01}}^{'} i\right) + (0 + 0)I$ where $I = \sqrt{i} = \sqrt{\gamma} \in F_{q^4}$ as can be remembered from section 2.2.3. So the line evaluation formula given above can be expressed as below:

$$l_{A,B}(Q) = \left(-\lambda_{A,B} \mathrm{z_A}^3 \mathrm{x'_Q} - \mathrm{x_A z_A} \lambda_{A,B} - \mathrm{z_C y_A}\right) + \left(\mathrm{z_A}^3 \mathrm{z_C y'_Q}\right)\mathrm{I}$$

Note that an element of F_{q^4} is represented as $A_0 + A_1 I$ where $A_0, A_1 \in F_{q^2}$.

2.2.6 Final Exponentiation

Final exponentiation in Step 9 of Algorithm 1, $f \leftarrow f^{(q^k-1)/r}$, can be reduced to two smaller hard exponentiations with the help of property described in [16]. Exponent $(q^4 - 1)/r$ is separated into two parts; $(q^2 - 1)$ and $(q^2 + 1)/r$. The method for performing the final exponentiation using these two parts is described below.

Let's write $f = F_0 + IF_1$ such that $F_0, F_1 \epsilon F_{q^2}$. We can handle the first exponent operations with $(q^2 - 1)$ as follows:

$$\begin{aligned} t = f^{q^2-1} &= (F_0 + IF_1)^{q^2-1} \\ &= (F_0 + IF_1)^{q^2}(F_0 + IF_1)^{-1} \\ &= \left(F_0 + I^{q^2} F_1\right)(F_0 + IF_1)^{-1} \\ &= (F_0 - IF_1)(F_0 + IF_1)^{-1} \text{ [16].} \end{aligned}$$

If we include the other exponent $(q^2 + 1)/r$ we obtain

$$f^{(q^4-1)/r} = t^{(q^2+1)/r} = t^{k_1 q + k_2},$$

where $k_1 = [(q^2 + 1)/r]/q$, $k_2 = [(q^2 + 1)/r] \, mod \, q$ and $t \epsilon F_{q^4}$, $t = (T_0 + IT_1)$ such that $T_0, T_1 \epsilon F_{q^2}$.

The first part of $t^{k_1 q + k_2}$ can be calculated as follows:

$$s = t^q = (T_0 + IT_1)^q = ({T_0}^q + I^q {T_1}^q)$$

$$= \left(t_{00}^q + i^q t_{01}^q\right) + I^q (t_{10}^q + i^q t_{11}^q)$$

$$= (t_{00} - it_{01}) + I^q (t_{10} - it_{11})$$

where $I^q = I^{q-1} I$ and $Fr = I^{q-1} = (I^2)^k = i^k \in F_{p^2}$.

$$s = (t_{00} - it_{01}) + Fr * (t_{10} - it_{11}) * I = S_0 + IS_1$$

Finally we have

$$f^{(q^4-1)/r} = s^{k_1} * t^{k_2}.$$

Two small exponentiations with exponents k_1 and k_2 are realized separately with basic binary exponentiation method [39] or using simultaneous exponentiation algorithm.

During calculation of variable t, one F_{q^4} inversion is computed. A F_{q^4} inversion can be reduced into F_{q^2} inversion and couple of multiplication in the subfield F_{q^2}. Since we use tower construction for extension fields, one inversion in F_{q^2}, in turn, can be written in terms of an inversion in F_q as described in Algorithm 6.

Algorithm 6: F_{q^2} Inversion Using F_q Inversion
Inputs: $a = a_0 - a_1 i,\ \ a_0, a_1 \epsilon F_p$ **Output:** $b = a^{-1}$, b = $b_0 + b_1 i$ 1. $t_1 = a_1 a_1$ 2. $t_2 = \beta t_1$ 3. $t_1 = a_0 a_0$ 4. $t_3 = t_1 - t_2$ 5. $t_4 = {t_3}^{-1}$ 6. $b_0 = a_0 t_4$ 7. $b_1 = -a_1 t_4$ - Total cost of the operation: $5F_q$ multiplication + $1F_q$ inversion + $2F_q$ addition

Finally, a F_{q^4} inversion is realized using a F_{q^2} inversion as described in Algorithm 7.

Algorithm 7: F_{q^4} Inversion Using F_{q^2} Inversion

Inputs: $A = A_0 + A_1 I;\ A_0, A_1 \in F_{q^2}$

Output: $B = A^{-1}, \mathrm{B} = \mathrm{B}_0 + \mathrm{B}_1\mathrm{I}$ such that $B_0, B_1 \in F_{q^2}$

1. $T_1 = A_1 A_1$
2. $T_2 = \gamma T_1$
3. $T_1 = A_0 A_0$
4. $T_3 = T_1 - T_2$
5. $T_4 = {T_3}^{-1}$
6. $B_0 = A_0 T_4$
7. $T_1 = -A_1$
8. $B_1 = T_1 T_4$

- Total cost of the operation: $5F_{q^2}$ multiplication + $1F_{q^2}$ inversion + $2F_{q^2}$ addition

In the following section hardware architecture of the design is explained.

3 Parametric and Compact Implementation of Hardware Coprocessor for Pairing on FPGA

Public key cryptosystems such as elliptic curve and pairing-based cryptography require complicated arithmetic operations. For example an $\boldsymbol{F_{q^2}}$ multiplication requires 3 $\boldsymbol{F_q}$ multiplications and several $\boldsymbol{F_q}$ additions and if the extension degree is increased, the operation becomes much more complex and time consuming. For instance, if the extension degree is increased to four, a multiplication on $\boldsymbol{F_{q^4}}$ requires 12 $\boldsymbol{F_q}$ multiplications. Since operations on extension fields are too complicated and require a diverse set of operations in the subfields, separate implementation of each operation in hardware may require prohibitively high logic area which makes software implementation preferable.

If implementation of these operations is realized on dedicated hardware, ASIC, for mostly speed concerns, it has some disadvantages compared to reconfigurable solutions as enumerated in the following:

1. Probably, the most problematic part of ASIC design is its cost; production of an ASIC design is many times more expensive than production of a design on FPGA, depending on the volume of the production.

2. Production of ASIC designs takes again much longer than making a design usable on FPGA (time-to-market factor).

3. Design and improvement of ASIC implementations takes much longer compared to the FPGA alternative.

4. If security levels or overall system architecture changes, ASIC no longer can be used and a new design and production are required.

5. As name implies, it is most of the time “dedicated” and cannot be used for any other purposes.

In this design, we make use of both the advantage of software and the flexibility of reconfigurable hardware designs. As indicated above, separate implementations of extension field operations may require too much logic area. Moreover, if the design choices change for some reasons (e.g. security, performance, compatibility), a specific unit computing a particular operation can become obsolete and requires re-design. For instance, if a dedicated unit is designed for solely F_{q^4} multiplication, and when the design is modified from F_{q^4} to F_{q^8} it becomes useless. Instead, we design and implement a programmable coprocessor on FPGA having basic arithmetic logic operation unit in its center, which is highly optimized for the target device. Thus, by changing the program of the coprocessor, many different operations can be performed on a simple reconfigurable hardware.

Our coprocessor is designed mainly for pairing operations; several pairing types such as Tate, Ate, on arbitrary elliptic curves can be calculated by simply changing the program of the processor. There is no need to change the hardware design. Other cryptographic calculations can also be implemented but some changes in state machines of the control circuit are needed. It is worth to note that, adapting processor for most of the other types of cryptographic applications can be realized by just modifying the control state machine, not all the design. So even if this processor is designed essentially for pairing operation, it can be adjusted for other applications with relatively small effort compared to designing it from the scratch. At this point we can see the advantages of this design over the pure software and ASIC solutions:

1. It is relatively cheap and easy to design, test and implement when compared to ASIC realizations.

2. It can be reconfigurable easily for different applications and design preferences whereas, ASIC cannot be.

3. It is much faster than software and it saves valuable CPU time for other operations

Performance and flexibility of the design is supported by the underlying architecture. General architecture of the design can be grouped into five parts: **i)**

arithmetic logic unit with modular inversion block, **ii)** fetch-decode-execute unit, **iii)** top controller, **iv)** program memory and **v)** data memory. Sub-modules of the processor can be briefly described as follows:

- **Arithmetic Core & Inversion Unit (ACIU):** Arithmetic core is composed of a modular adder-subtractor unit and a Montgomery multiplier unit. We have two arithmetic cores in the design, which makes our processor dual-core. In addition, we have a single inversion unit that computes multiplicative inverses in F_q.

- **Fetch-Decode-Execute Unit (FDEU):** This unit controls the program memory, data memory and ACIU. Since we have two arithmetic cores, to make it fully parallel we used two FDEU, each of which is connected to one ACIU.

- **Top Controller**: Top controller is the state machine that defines the characteristic of the processor. Namely, we design our top controller specifically for pairing operation. And for other kinds of applications it has to be modified but except for this module, other parts can still be used without any modification. There exists only one top controller that controls all the sub-modules.

- **Program Memory**: As the name indicates, this is the memory where the program code (it will be referred as "micro-code") is stored. To make use of full parallelism, this part is also instantiated for each FDEU.

- **Data Memory**: This part constitutes register banks used to store program data. Similar to program memory, each FDEU has its own data memory. Naturally, data transfers among FDEUs are allowed to combine the result of each core to a single one.

To summarize, a specialized Top Controller unit commands FDEUs to execute a block program (can be referred as function or micro-code henceforth), then FDEUs fetch that program (micro)-instructions from the program memory. After instructions are fetched, the FDEUs parse the instructions into three parts: Operation type, address of the operands and the address of which the operation result will be written. Then FDEU fetches the required data from the data memory and commands ACIU to execute

a given operation with given data. When ACIU finishes its operation, the result is sent to the data memory area which is pointed by the instruction. General overview of the co-processor architecture is depicted in Figure 3.

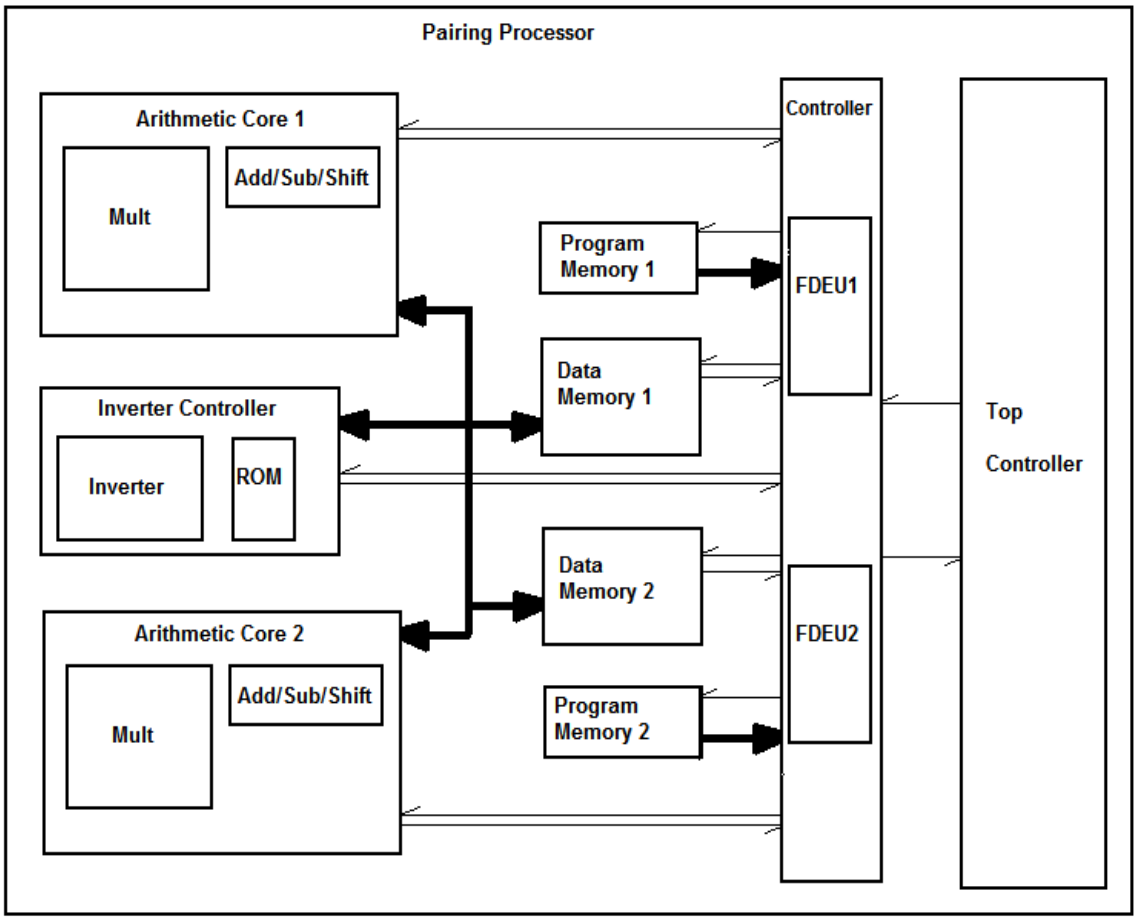

Figure 3: General Overview of the Processor Architecture

In Figure 3, bold lines represent a bus which carries both data bits and the control signals, the thin lines denotes buses that carry only control signals.

There are some other points that need further explanation. Firstly, inverter is capsulated into a controller and it has a separate program ROM. Secondly, FDEUs are encapsulated into a single controller unit. And eventually, FDEUs do not have direct connection with the data memory; instead they only have control lines. These are adopted design preferences here and will be clarified in the subsequent chapters.

Top controller informs the controller to execute a specific block of micro-code (BMC). If a function is parallelizable and if it is coded in a parallel manner, then for that function there will be a micro-code in both of the program memories for the corresponding block of the code. So when commands come from top controller, FDEUs

read the corresponding BMC from the program ROM and begin to execute them in a parallel manner. Program memory is filled by the programmer with micro-code of the processor before the execution. Data input is transferred to the controller via program memory. The controller does not directly handle the input and the output data to ACIUs; rather it handles the switching of data. Thus, ACIU can read and write data directly, which is much faster. This method can be considered as micro-DMA (direct memory access). We add the word “micro” because in normal DMA, the processor does not participate in data transfer and thus can deal with other processes while peripherals makes memory access faster. But in our case, the data transfer happens within the processor, hence the name micro-DMA. Yet, if we consider the controller as a processor, we can say that this method makes data access faster for ACIUs through a shortcut to the data memory. Although controller starts the transfer operation, it does not carry data blocks over itself to ACIU.

After the execution is performed by the ACIUs, the controller returns a done signal to the top controller which becomes ready for the next functions. At the end of all operations, top controller raises a finish flag and result is accessible in the predetermined address of the data memory. Last, but not least point is about the choice of RAM type to implement data memory. It is designed as dual port RAM, which allows simultaneous read operations. The reason that makes this choice is important is related to the arithmetic core. Since, the arithmetic cores use two operands for most operations, such as multiplication, addition and subtraction, it requires that operands be ready at both input ports at the same time. So, dual port RAM is a compact solution for the requirement.

ACIU includes multiplier, adder/subtractor/shifter and inverter. In fact, multiplier and inverter unit could be implemented by using just adder/subtractor/shifter unit, but this case would have two drawbacks. First, writing a program for the processor for this purpose would be too complicated. Secondly, total execution time for a cryptographic operation would be affected badly although working frequency does not change. What makes the execution time longer is that the control mechanism and data transfer cycle at each step of the multiplication and inversion algorithms. Especially, when we consider that the size of operands is around hundreds of bits, the effort needed for data transfer cycles would be prohibitively high. Because of this reason, multiplication and inversion

units are realized in hardware in highly optimized fashion. Detailed description of each sub module is explained in the following sections.

3.1 Arithmetic Core & Inversion Unit

ACIU composed of two main parts: arithmetic cores and inversion controller. These two parts are detailed in following two sub sections.

3.1.1 Arithmetic Core

Arithmetic core composed of two components: multiplier and adder/subtractor/ shifter block. Instead of making a separate shifter, adder is also used as a shifter. Before going further into details of the sub modules, we explain general overview of the arithmetic core itself and I/O signals that belong to the core.

Arithmetic core is designed to present a user friendly interface to the controller and to gain speedup during the most common instructions. Arithmetic core accommodates a state machine between the controller and operation units as an interface. With the help of this interface, the controller can communicate easily and efficiently with the arithmetic cores. The controller interacts with the arithmetic cores using commands which also constitute the opcode part of the micro code. The controller sends its command and then adjusts the data memory address to provide the appropriate operands to the arithmetic cores. After the result is found, data is written to indicated part of the data memory directly by the arithmetic core, which also sends a signal to the controller to indicate that operation is completed. I/O interface of the arithmetic core is depicted in Figure 4.

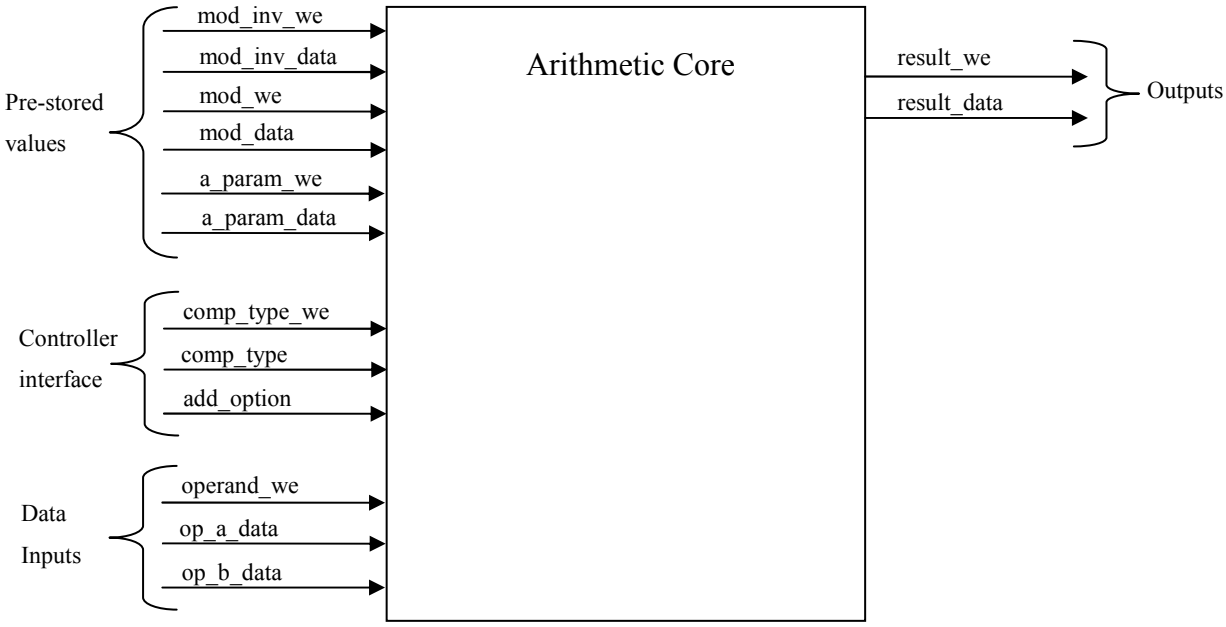

Figure 4: Arithmetic Core I/O Interface

Pre-stored values that are constant and frequently used during the operations should be stored in the arithmetic core before the execution of any operation. Therefore, pre-stores values are kept in a RAM inside of the arithmetic core. Storing these values inside the arithmetic core reduces the load over the controller per operation. For example, the controller is saved from loading the modulus at every operation.

As can be seen in Figure 4, the controller interface is pretty simple; controller should load the “opcode” along with an “enable” signal. This signal takes one clock period. An additional input is used to tell the arithmetic core whether to make addition or subtraction.

Data is fed into the core directly from the data memory. Decision about the location of data is made by the controller. After opcode is stored in the core, operands come from the data memory with the help of switching of the controller. Finally, when the result is produced it is directly written in the data memory by the core. Again the data bus switched to the correct location by the controller just after the loading of opcode. Only a signal is sent to the controller to indicate that the operation is finished.

Determining the possible opcodes is the focal point in the arithmetic core design. The operations that are performed often are made an opcode. For example, if squaring is executed many times or if it might be used frequently in other applications, then squaring can be made an opcode. Thus, we get rid of read and write cycles every time in

certain and commonly used operations. Again, there is a trade-off between time and area. If this kind of optimization were not made in the opcode selection, the core would be much simpler. However we prefer to improve the execution time of the overall system. Below a list of opcodes is given in Table 2.

Opcode	Definition
0000	Idle: Do nothing
0001	$c = a \times b$: Regular multiplication
0010	$c = a + b$: Regular addition
0011	$c = a + a$: Regular shift left once
0100	$c = 3 * (a * b)$: First $a \times b$, then two addition
0101	$a^2 \times b$, $b^2 \times a$, a^3
0110	$A * a * b$ A is elliptic curve coefficient a in $y^2 = x^3 + ax + b$
0111	$c = 2 * a * b$: Multiply then add

Table 2: Opcodes and their Definitions for Arithmetic Core

Arithmetic core can take a new opcode when it finishes the current operation. It is not allowed to load new opcode during an operation. Operands are fed into the core word by word and all the sub modules process the data on word-basis. Size of words is changeable. As can be seen from the Table 2, four bits are reserved for the opcode while most significant bit is always zero. This is optional part. Opcodes can be extended with some changes in the core logic for different kinds of applications. Detailed explanation about the building blocks of the core is given in the following subsections.

3.1.1.1 Multiplication Module

Since Montgomery [40] offered one of the most efficient methods for hardware multiplication, this block is designed to implement Montgomery multiplication. Selection of this block as a Montgomery multiplier (MM), affects also the choice of inverter and the way data is given to circuit. The MM produce a result for given inputs "a" and "b", $MM(a, b, M) = a \times b \times R^{-1} \ (mod\ M)$, where R is constant and usually

chosen as $R = 2^{modulus\ bit\ length}$. For this reason, our arithmetic units accepts operands in Montgomery domain (i.e. $a \times R$ instead of just a,) to avoid information loss.

The MM is taken from a previous work [41]. It is a generic architecture and uses Coarsely Integrated Operand Scanning (CIOS) Montgomery Multiplication algorithm:

Algorithm 8: CIOS Montgomery Multiplication Method [42]

Inputs: a[j], b[j]: $j_{th.}$ word of operands (w bits each)
M[j]: $j_{th.}$ word of modulus (w bits each)
k: Number of words in the operands and modulus
W: 2^w, C: carry, S: sum
$M[0]^{-1}$: multiplicative inverse[1] of M[0]
|| : used for concatenation
t := 0

Output: t[i]:= intermediate and final results

```
 1. for i=0 to k-1
 2.   C :=0
 3.   for j=0 to k-1
 4.     C||S := t[j] + a[j] × b[i] + C
 5.     t[j] := S
 6.   C||S   := t[k] + C
 7.   t[k]   := S
 8.   t[k+1] := C
 9.   C      := 0
10.   z      := t[0] × (-M[0]^-1) mod W
11.   C||S := t[0] + M[0] × z
12.   for j=1 to k-1
13.     C||S   := t[j] + M[j] × z + C
14.     t[j-1] := S
15.   C||S   := t[k] + C
16.   t[k-1] := S
17.   t[k]   := t[k+1] + C
```

The CIOS method is a word based method as can be seen in Algorithm 8. It takes and processes the operands word by word and it forms the result in the same

[1] "Least significant word of inverse M" in mod 2^r, where $2^{r-1} < M < 2^r$

manner. The CIOS is one of the most efficient algorithms for implementing the MM on FPGA [42].

Algorithm 8 is implemented in a pipelined manner to take advantage of parallelism and to speed up the design in FPGA. It makes use of hardwired multiplier blocks in the FPGA thus, both gains from logic area and provides acceleration in multiplication operations. The design is very flexible and parametric. Number of pipeline stages, number of bits in each word and number of words in operands can be adjustable. Therefore it provides time-area trade-off. For example, area can be reduced by decreasing both the number of pipeline stages and the number of bits a word. In return, the total execution time increases. This kind of design is very helpful for adjusting overall time-area trade-off because multiplication is the most commonly used operation in the pairing. Changes in its timing characteristics affect total timing of the application in a substantial manner. In Figure 5, I/O interface of the multiplication unti is given:

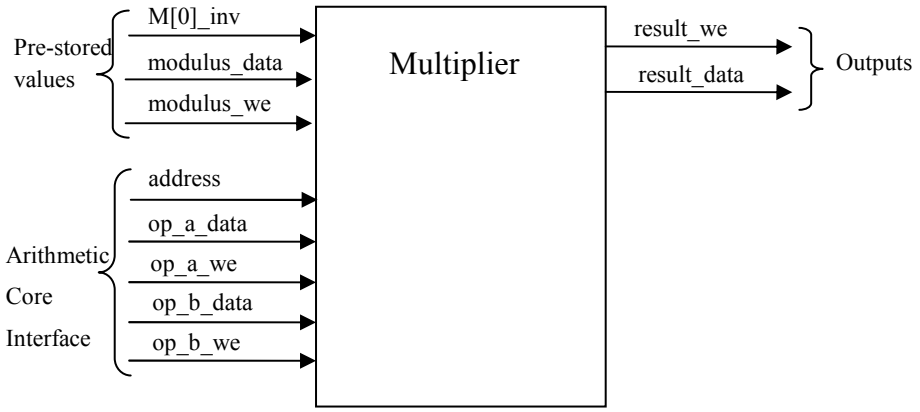

Figure 5: Montgomery Multiplier I/O Interface

Pre-stored values are stored into the module before the calculation starts. Each input operand has separate load inputs. Thus, for some operations as indicated in the opcode, such as “0101”, as one operand from previous calculation remains intact, the other operand takes a new value. Address input specifies the address in the RAM where the operands are stored. This input is automatically increased by the arithmetic core. When calculation finishes, the result becomes available with an active write enable signal. Output of the multiplier is switched as indicated in the opcode by the core.

3.1.1.2 Addition/Subtraction/Shifter Module

All three modules (i.e. adder, subtractor and shifter) are realized in the same unit. Although it is normal to have adder and subtractor together, we prefer to implement shifter using addition. This has two reasons: Firstly, it makes the control circuit simpler, secondly even if we use normal shifter, we need a modular adder since we must guarantee that after shift operation result may have to be reduced since this is a modular shift operation (i.e. shift operation in F_q). Inner structure of the modular adder is given in Figure 6.

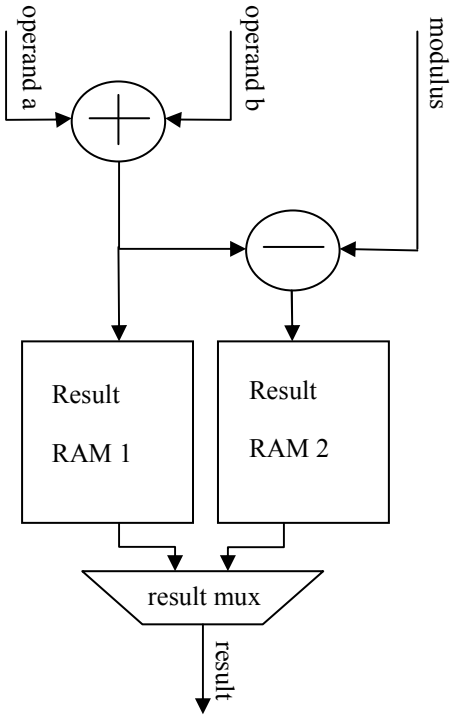

Figure 6: Modular Addition Architecture

Figure 6 is an illustration of modular addition, and in case of the subtraction operation, the adder (+) and subtractor (-) units are switched. Operation immediately starts with the load of operands. Operands are added and stored into RAM 1 and modulus is subtracted from the addition of the operands and stored into the RAM 2. Both operands and the modulus fed into the module word by word, and carry-out from the previous word is fed into the next word as carry-in. After all the words are exhausted, carry-out of the last word is examined. If the result of subtraction with the modulus M (i.e. $a + b - M$) is negative then, it is concluded that $a + b < M$ and result in

the RAM 1 is valid and vice versa otherwise. The reason of using two RAMs is providing the result immediately after the data load finishes.

We use registers between the two adders to break the long delay in the critical path. Other than that we do not make anything to speed up the adders. This is due to the fact that, while synthesizing the adder the router automatically places the logic around the fast carry propagate lines hence, it does not create a bottleneck for the design.

Adder is also parametric like the other parts of the design. Input word bit length and the total input word number can be adjusted. I/O interface of the adder is shown in Figure 7.

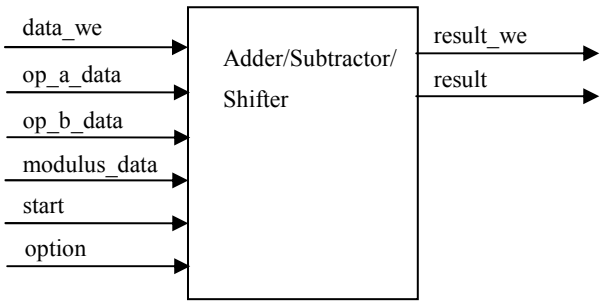

Figure 7: Modular Addition I/O Interface

The “option” input is used to alternate from subtractor to adder; the “start” input is used to start the output stream. This input is utilized to postpone the output in case of such a need occurs in timing adjustment. To start an operation, the modulus and the operand data should be given with an active write enable signal. For every operation modulus data is fed into the module and start signal is adjusted automatically by the core.

3.1.2 Inverter Controller

Inversion is used in the final exponentiation part of the pairing. Although it is not used many times, it plays a critical role to reduce the execution time spent on the final exponentiation. The execution time of extension field exponentiation is halved if a dedicated inversion unit is used. We prefer to add an inverter unit to make use of the

advantage it brings in timing with the cost of some logic area. Since inversion is a rare operation, instead of inserting it to the arithmetic core we build a single inversion unit.

During the execution of a functional block for inversion operation, control of the data memory is left to the inverter controller. Inverter controller possesses its own internal program ROM. Every time it is notified by the controller, it executes the micro code coming up next. It reads data and writes the result to the specified addresses.

The structure of the inverter controller is highly simple. It has a state machine to control the read/write operations to the data memory and it switches data in and out of the inversion. The I/O interface of the inverter controller is given together with inner block diagram in Figure 8.

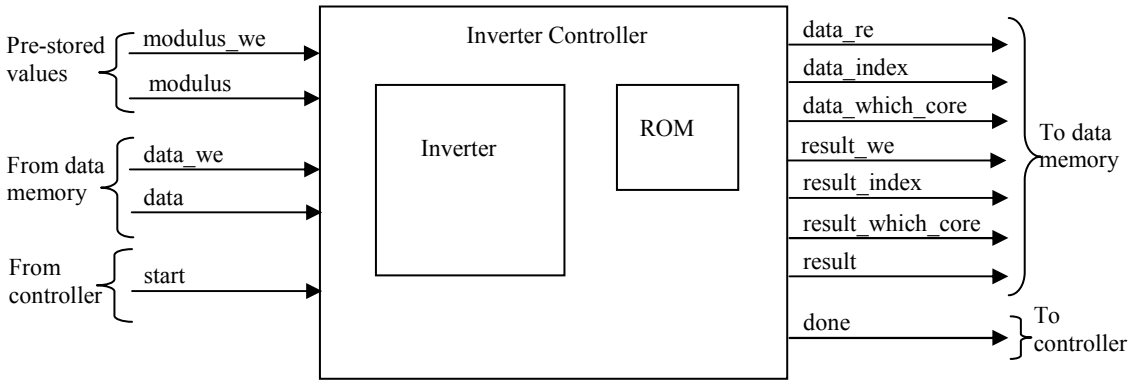

Figure 8: Inverter Controller I/O Interface

Inverter controller starts operation with the active “start” signal from the controller. It reads micro code from the ROM and sends the address of data to be read to the data memory. After the result is obtained it is written to the data memory, then the controller is sent a an active “done” signal.

Inverter controller has two sub modules; ROM and inverter. ROM is either synthesized with “Block RAM (BRAM)”, which is a hardwired RAM in the FPGA, or is synthesized using “Distributed RAM”, which is a RAM constructed of look-up tables (logic elements) in the FPGA. We picked distributed RAM since, usually work load of the inverter is low and thus it does not need a large memory to store the micro code. Certainly, inversion circuit is the most important part. It is detailed in the following subsection.

3.1.2.1 Montgomery Inverter Module

Modular inversion is widely used in cryptographic applications particularly in public-key crypto-system. For instance, during the calculation of private key in Digital Signature Standard [43] and in RSA [44] and in elliptic curve cryptography, [46]-[48], it is commonly used. For this reason, inversion module is a must for not just pairing operation but also for a cryptographic processor. Thus, we design a low area and parametric modular inverter optimized for FPGA devices. During the design, to eliminate the delays of long carry chains, we favor the word-based design and move away from full precision adders and subtractors. We process the data similar to a general-purpose computer, on word-basis and we make use of the advantage of the FPGA. We use the Montgomery modular inversion algorithm to make it compatible with the Montgomery multiplier. We tested efficiency of the design on Xilinx Spartan-6 FPGA, which is a new generation of low cost FPGA with low power consumption. Implementation results show the design reaches frequencies higher than 200 MHz with a few hundreds of logic resources.

Modular inverse on modulo M, where M is an odd prime, is defined as follows: If $z, x \in [1, M-1]$ and $z * x \equiv 1 \ (mod\ M)$ then, z and x are multiplicative inverse of each other in modulo M. It is displayed as $x \equiv z^{-1} \ (mod\ M)$ [50]. Montgomery inverse algorithm is first proposed by Kaliski [49], which works fine with MM.

Generally inversion is used together with other arithmetic operations in cryptography. By looking at its usage ratios in overall application, satisfying high working frequency with high area usage is not acceptable for the inverter design. Since, it is usually not timing bottleneck for most cryptographic applications. On the other hand, its working frequency should not be lower than the multiplier unit in order not to reduce the entire working frequency of the system. As a result, the aims are a low area and a fast inverter module.

In this design, input is handled as an array of words instead of full precision, where each word is taken into operation at every clock cycle. Also the output is produced word by word. Benefit of using word-based implementation is apparent in size of the registers and adder/subtractors used. While keeping the size of the circuit small, it also raises the frequency with reduced carry-chain lengths in the adders.

Besides this, we can make use of BRAMs for storage of variables in place of registers. In the same manner, this choice reduces the total logic source usage of the inverter circuit.

Our design has some superiority over the full precision design. Despite the fact that full precision design can complete the overall calculation in a shorter time, it consumes vast amount of logic resources and achieves very low working frequencies. This leaves a very limited area to realize essential part of the design. In addition to this, low frequencies create a need for another clock source to drive the lower frequency part of the circuit. Without doubt, this will result in a more complex and a hard to handle design. Our design demonstrates that, the word-based architecture covers more than 10 times less logic resources, moreover it can work at 3 to 4 times higher clock frequencies than half precision structures. We realize a parametric design that can be re-synthesizable for any word length and input size. Thus, parametric design provides flexibility to meet the implementation constraints. We describe the Montgomery inversion algorithm in detail in the following section.

3.1.2.1.1 Montgomery Modular Inversion Algorithm

Let M be an odd integer and a is an integer such that, $a \in [1, M-1]$ and if the equation holds: $a \times x \equiv 1(mod\ M)$. It can be expressed as below:

$$x := ModInv(a) \equiv a^{-1}(mod\ M).$$

It is important to note that multiplicative inverse of a only exists if a and M are coprime.

A Montgomery modular inverse algorithm is proposed by Kaliski based on extended binary GCD approach [49], [51]. The Montgomery multiplicative inverse of an integer is expressed as follows:

$$X := MonInv(a * 2^m) = a^{-1} * 2^m\ (mod\ M).$$

Here m stands for bit length of the modulus M. The Montgomery inversion algorithm is composed of two main phases. These phases are described in Algorithm 9 and Algorithm 10, respectively.

Algorithm 9: AlmMonInv(*a*, *M*) (Phase I) [49]

Inputs: $a \in [1, M-1]$
M: modulus
$u \leftarrow M, v \leftarrow a, r \leftarrow 0, s \leftarrow 1, k \leftarrow 0$
Output: r and k, satisfied that $r = a^{-1} \times 2^k \ (mod\ M)$ and $m \leq k \leq 2m$

```
1. while v > 0 do
2.   if u is even then
3.     u ← u/2, s ← 2 × s
4.   else if v is even then
5.     v ← v/2, r ← 2 × r
6.   else if u > v then
7.     u ← (u−v)/2, r ← r + s, s ← 2 × s
8.   else u ≤ v
9.     v ← (v−u)/2, s ← r + s, r ← 2 × r
10.  end if
11.  k ← k + 1
12. end while
13. if r ≥ M then
14.   r ← r − M
15. end if;
16. return M − r, k
```

The output of the "Algorithm 9" is described as $r := AlmMonInv(a, M) = a^{-1} \times 2^k \ (mod\ M)$, where $m \leq k \leq 2m$. The output of the first phase is also known as the almost Montgomery inverse of a with respect to modulus M. If *NW* stands for number of words in the modulus then our design takes *NW+4* clock cycles to complete one iteration of the *while loop*. Since phase I is iterated *k* times, then it takes totally $k \times (NW + 4)$ clock cycles to complete the loop. After while loop, two more iterations are needed to calculate the remaining steps. Thus, it takes $(k + 2) \times (NW + 4)$ clock cycles to finish the phase I of the inversion. The result of the first phase is then transformed to the Montgomery domain by several iterations in the second phase of the algorithm, which is depicted in Algorithm 10.

Algorithm 10: MonInv(r, M, k) (Phase II) [49]

Inputs: r, M and k from AlmMonInv
Output: x, satisfiying $x = a^{-1} \times 2^{2m} \ (mod\ M)$

```
1. for i = 1 to (2m − k) do
2.     r ← r ≪ 1
3.     if r ≥ M then
4.         r ← r − M
5.     end if
6. end for
7. return x ← r
```

As can be seen phase II takes $(2m - k)$ iterations. The number of iterations depends on the value *k,* which is the output of the first phase. However, total number of iterations in phase I and phase II is constant and it is $2m$. As a result, total processing time of the implementation is about $(2m + 2) * (NW + 4)$. Each iteration in phase II does not take $NW + 4$ periods, but sometimes it takes $NW + 3$ clock cycles. Therefore $2m * (NW + 4)$ clock cycles will be a more accurate formula to approximate the execution time of the inversion operation.

To summarize, if an integer a is given in Montgomery domain, i.e. $a \times 2^m \ (mod\ M)$, the output of the first phase is:

$$AlmMonInv(a \times 2^m, M) = \ (a \times 2^m)^{-1} \times 2^k \equiv \ a^{-1} \times 2^{k-m} \ (mod\ M).$$

The second phase clearly multiplies the result with $2^{2m-k} \ (mod\ M)$ to put it back in the Montgomery domain as follows:

$$a^{-1} \times 2^{k-m} \times 2^{2m-k} \ (mod\ M) \equiv a^{-1} \times 2^m \ (mod\ M).$$

If the result is needed in the normal domain, i.e. a transformation from $a^{-1} \times 2^m$ to a^{-1} is required, it suffices to multiply the result by 1, using a Mongomery multiplier. Consequently, we obtain got $MM(a^{-1} \times 2^m, 1) = \ a^{-1} \times 2^m \times 1 \times 2^{-m} \equiv a^{-1} \ (mod\ M)$. In the following section, inner structure of the inverter is explained.

3.1.2.1.2 Montgomery Inverter Architecture

Proposed inverter has a parametric design, in the sense that word size can be adjusted to scale the architecture into desired size or frequency requirements. In other words, we offer to operate on changeable word size instead of operating on all or half of the input bit size as proposed in [52] according to the design criteria. Thus, we provide an obvious increase in operating frequency. Among the factors to accelerate the design is reduced carry-chain length in adders/subtractors and fewer number of connections between CLBs. In addition, reduced word size also decreases the register and LUT usage resulting in lower area consumption. Furthermore, logic area usage is decreased more by using BRAMs for the storage of variables during operation. Nonetheless, our design is synthesizable by using either distributed RAM or BRAMs. Implementation results with and without BRAMs are given in the further sections.

Most time consuming part of the operation is the first phase of the algorithm (Algorithm 9) that dictates the main structure of the architecture. Second phase of the algorithm (Algorithm 10) is accomplished over the same architecture by adding some signals for control purpose. The *while loop* in Algorithm 9 is the most dominant segment in the first phase, hence the architecture is mainly shaped depending on this segment. All data inputs of all modules are size of WL, where WL stands for desired bit length for a word.

Inverter is realized using multiplexers, adders/subtractors and RAMs to store the variables. RAMs operate like FIFO (first-in first-out) structure. Firstly, least significant word (LSW) is stored in a RAM. While reading the RAM, first LSW of the variable is read back and finally most significant word (MSW) of the variable is obtained. Operation is controlled using select inputs of multiplexers and read/write enables of RAMs. First part of the algorithm is separated into two parts: i) **u/v** and ii) **r/s**. These parts are depicted Figure 9 and Figure 10, respectively. Execution of the first part of the algorithm is influenced depending on whether "*u*" or "*v*" is even or they are both odd numbers.

Before the execution of the first phase, the modulus value is stored in the "RAM_M" and operation starts with data load signal, which is named as "Inv_in_we". When this signal is received, the modulus is read and stored in the "RAM u" as its

initial value. The values of zero and one is stored to the “RAM r” and “RAM s”, respectively as initial values. After loading initial values, while loop starts with “r_sig”, which is the read signal of RAMs. Every iteration of the “while loop” (and “for loop”, in the second phase) is performed with “r_sig” signal. Write enable signals “w_sig0” to “w_sig2” are sequentially shifted version of the signal “r_sig”. Proper write enable signal is chosen according to the delay of data path. All RAMs used in the design are dual-port RAMs, which allow simultaneous read and write operations from different addresses. Thus at each iteration of Algorithm 9 (steps 1 - 12), RAM read/write operations can be performed in less clock cycles compared to the case of using single port RAMs.

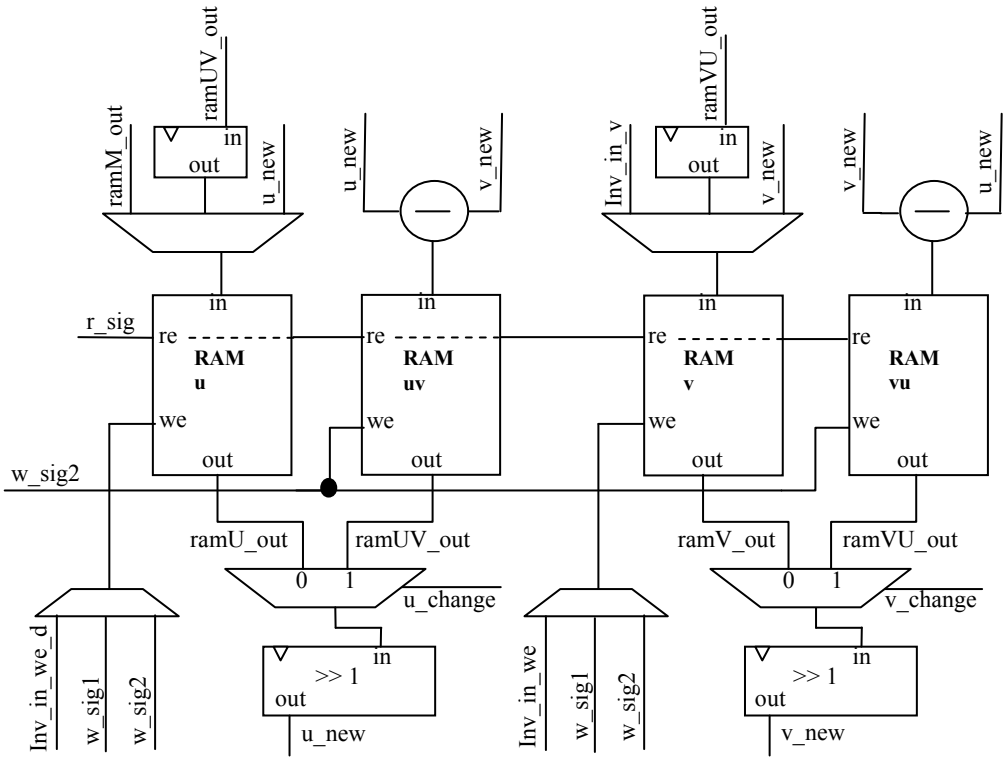

Figure 9: U/V Part of the Inverter

While reading data with “r_sig” signal, the least significant bit is examined to decide if the number is odd or even. If one of the numbers (*u* or *v*) is even, then after the read of first word of the variables we can decide to execute upper half of the while loop. But if none of the numbers are even, we should compare them to determine which is greater. After this “u>v” or “v>u” part of the while loop is executed.

However, deciding which number is greater than other is not an as easy as deciding whether they are even or not since it is not sufficient to examine just the LSW of a variable. Instead, all words of the variables are needed. For instance, it is required first to read all words of the variables to decide which one smaller. Then, reading them again to subtract the smaller from larger is overly time-consuming process. In that case, for the one iteration of the while loop, two read operations have to be performed. To eliminate these overly complicated operations, we add one more subtractor to the circuit. If both numbers are odd, then "u-v" and "v-u" operations are performed in parallel and result of each is stored in two separate RAMs (i.e. RAM uv or RAM vu) . Afterwards, these two results are checked. Naturally, the positive result is valid which is used as the new value of either "*u*" or "*v*". The valid data is selected using "u_change" and "v_change" signals. For instance, if valid "*v*" value is "RAM vu" for now, then "v_change" value is set to '1' and it is set to '0' vice versa. If upper half of the while loop have to be executed, then "*u*/2" or "*v*/2" values are stored in either "RAM u" or "RAM v". "u_new" and "v_new" lines stand for "*u*/2" and "*v*/2" values. Current values in RAMs are processed in one right shifter unit, which divides the current values by 2. We perform $u/2 - v/2$ instead of $(u - v)/2$ for the lower half of the while loop. But this does not cause a data loss, since in the lower part of the loop both numbers are odd. There is an important point to note. "RAM uv" and "RAM vu" are continuously filled with new difference values of "*u*-*v*" and "*v*-*u*" respectively. Therefore, the valid data is has to be copied to the other RAM. "ramUV_out" and "ramVU_out" signals are added to the input of "RAM u" and "RAM v" for that reason.

Loop in the first part of the algorithm ends when the value "*v*" reaches to zero. If conditional structure in the algorithm is carefully examined, it can be seen that algorithm only terminates after *u* is subtracted from *v* (therefore *u* is smaller or equal). Therefore, input of the "RAM vu" is connected to a zero comparator unit. Zero comparator units detects zero value and raises a flag when zero value is stored to the RAM to terminate the execution of the while loop.

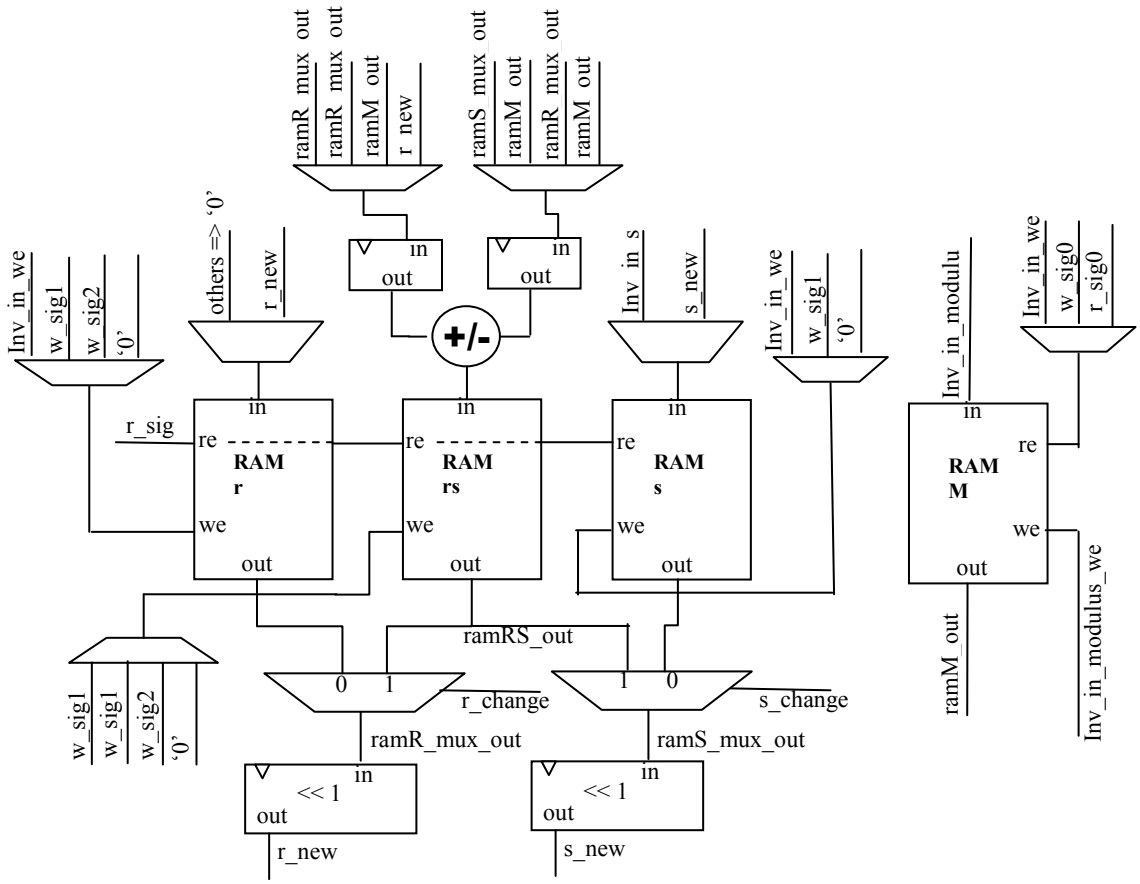

Figure 10: R/S Part of the Inverter

Variables "*r*" and "*s*" change depending on the values of "*u*" and "*v*" during the execution of the while loop. If upper half of the while loop executes, new values of "*r*" and "*s*", "r_new" and "s_new", are processed by one-bit left shifter (multiplier by 2). "RAM r" and "RAM s" are continuously updated with shifted values of "*r*" and "*s*". "RAM rs" stores the $r + s$ value when lower half of the while loop executes. After deciding whether u or v bigger than the other, value stored in "RAM rs" updates either *u* or *v*. If the value stored in "RAM rs" is used to update the variable r, "r_change" flag is set to one. The same rule applies for "s_change" flag. The adder/subtractor in the input of the "RAM rs" operate as an adder for the while loop and as subtractor afterwards. Two registers are put in front of the adder/subtractor to reduce the critical path delay. Second phase of the inversion operation (Algorithm 10) is realized using "RAM rs", "RAM r" and "RAM M". At each iteration of the "for" loop in the second phase the shifted value of r is stored to "RAM r", and the value $2 \times r - M$ is stored to the "RAM rs". If stored value to the "RAM rs" is negative then valid r value is in "RAM r" and in "RAM rs", otherwise.

We use dual port RAMs, which enables to read and write at the same time, despite that one iteration takes $(NW + 4)$ clock cycles. The delay of four clock cycles are due to the following: RAMs are synchronous read/write RAMs, thus when enable signal rises, data to read becomes available at the output after one cycle. One cycle is lost at the shifter modules and one at the input registers of the adder/subtractor. One more cycle is consumed after the subtraction. Deciding whether the result is negative or not, occurs after subtraction taken place. Thus in total, we spend four more cycles additionally. Implementation results of the design are given in the followings.

3.1.2.1.3 Implementation Results of the Inverter Unit and Other Metrics

Implementation results are given using ISE Design Suite 12.1 for target device Spartan-6SX45T FPGA (XC6SLX45T-3FGG484). Synthesizer presents two optimization methods for the design, one of them is speed and the other one is area optimization. In addition to these, we placed and routed the design utilizing both BRAM and distributed RAM to see how they affect area and speed results of the implementation. Spartan-6 FPGAs presents hardwired adder/subtractor modules; nevertheless we prefer not to use them. It have several reasons. Firstly the provided carry-chain lines are fast enough. And secondly since our design is word based, not much logic source is spent for adders. And lastly, it provides compatibility with other FPGAs.

Register and LUT usage are metrics for area coverage. If BRAM are used, additional constant eight BRAMs are added to the resource list. Logic resource usage is directly proportional to the chosen WL. This is an expected result since WL increases sizes of multiplexers, adders and registers. However achievable maximum frequency is not directly related with WL until a certain word size is reached. Negative effect of WL over frequency begins to appear after around 40 bits of word size. A suitable value for WL can be chosen for given design constraints, i.e. area usage, maximum working frequency and total completion time. For total execution time, previously given formula can be used: $2m * (NW + 4) * clock\ period = 2m * (NW + 4) * 1/MF$. Best area result is obtained under area optimization preference and using BRAM. Best speed result is obtained using under speed optimization preference using distributed RAM. All results are obtained from ISE Design Suite 12.1 after PAR process. Results are

enumerated in Tables 3, 4, 5, and 6, where m, WL, REG, LUT, MF, T, and TA denotes the bit size, word length, register usage, LUT usage, maximum clock frequency achieved, execution time, and time-area product, respectively.

m	WL	REG	LUT	MF	T (us)	TA
64	8	233	266	199,46	7,70	2,05
64	16	360	376	188,22	5,44	2,05
64	32	631	643	181,99	4,22	2,71
128	8	251	277	191,25	26,77	7,42
128	16	378	389	185,47	16,56	6,44
128	32	649	652	170,22	12,03	7,84
256	8	269	296	188,24	97,92	28,98
256	16	396	399	178,43	57,39	22,90
256	32	667	665	168,49	36,47	24,25
512	8	288	352	185,33	375,72	132,25
512	16	414	420	175,86	209,62	88,04
512	32	685	674	166,87	122,73	82,72
1024	8	306	519	184,19	1467,70	761,74
1024	16	433	522	174,19	799,49	417,34
1024	32	703	695	165,32	445,97	309,95

Table 3: PAR Results Using Distributed RAM Under Area Optimization

m	WL	REG	LUT	MF	T (us)	TA
64	8	242	305	243,64	6,30	1,92
64	16	371	413	229,79	4,46	1,84
64	32	657	693	252,24	3,04	2,11
128	8	264	329	234,1	21,87	7,20
128	16	390	433	235,01	13,07	5,66
128	32	679	698	252,24	8,12	5,67
256	8	287	359	232,66	79,22	28,44
256	16	404	436	216,14	47,38	20,66
256	32	678	696	217,92	28,19	19,62
512	8	289	399	191,04	364,49	145,43
512	16	432	476	228,75	161,15	76,71
512	32	698	715	216,47	94,61	67,65
1024	8	329	650	231,08	1169,88	760,42
1024	16	444	587	222,11	627,00	368,05
1024	32	723	738	223,15	330,40	243,83

Table 4: PAR Results Using Distributed RAM Under Speed Optimization

m	WL	REG	LUT	MF	T (us)	TA
64	8	170	202	200,04	7,68	1,55
64	16	233	282	192,37	5,32	1,50
64	32	376	424	199,53	3,85	1,63
128	8	188	212	193,51	26,46	5,61
128	16	251	295	189,18	16,24	4,79
128	32	394	462	172,67	11,86	5,48
256	8	206	235	193,51	95,25	22,38
256	16	269	304	181,77	56,33	17,13
256	32	412	476	170,82	35,97	17,12
512	8	224	259	190,19	366,12	94,82
512	16	287	323	179	205,94	66,52
512	32	430	485	169,08	121,13	58,75
1024	8	242	286	192,75	1402,52	401,12
1024	16	305	347	173,19	804,11	279,03
1024	32	448	506	167,44	440,32	222,80

Table 5: PAR Results Using BRAM Under Area Optimization

m	WL	REG	LUT	MF	T (us)	TA
64	8	175	233	204,43	7,51	1,75
64	16	237	313	204,43	5,01	1,57
64	32	379	479	204,43	3,76	1,80
128	8	189	236	204,43	25,05	5,91
128	16	253	330	204,43	15,03	4,96
128	32	398	501	204,43	10,02	5,02
256	8	207	259	204,43	90,16	23,35
256	16	274	334	204,43	50,09	16,73
256	32	418	523	204,43	30,05	15,72
512	8	224	287	190,76	365,02	104,76
512	16	287	336	192,18	191,82	64,45
512	32	437	530	204,43	100,18	53,10
1024	8	242	308	194,36	1390,90	428,40
1024	16	307	376	187,58	742,42	279,15
1024	32	456	571	204,43	360,65	205,93

Table 6: PAR Results Using BRAM Under Speed Optimization

Maximum frequency results are given in terms of MHz. Total working time is calculated using the formula $2m * (NW + 4) * 1/MF$. Time-area product is calculated using the formula $LUT * T/1000$. As one can observe from the tables showing the PAR results using distributed RAM (i.e. Tables 3 and 5), surprisingly

maximum achievable frequency remains almost unchanged with the word length. This is because of delay in the adder until around word size of 40 bits does not exceed the delay of the state machine implementing the control circuit. Small fluctuations in maximum frequency are due to the changes in path delay caused by checking for total word count.

On the other hand, PAR results using BRAM shows less fluctuations in maximum clock frequency especially for speed optimization case (Table 6). This is due to the fact that there is a bigger delay incurred in the output of BRAM. The BRAMs have constant locations inside the FPGA, which are usually in the middle of the FPGA. And therefore, when the main part of the circuit is placed it generally becomes far away from BRAMs. There occurs a large net delay, which exceeds any inner delay caused by the controller or adder circuit. To overcome this situation, outputs and inputs of the BRAM should be registered which adds two more cycle of latency to usual read/write cycle of RAM. This approach may increase the maximum working frequency but on the other hand it also increases total execution time.

We compared our design with a previous implementation that utilizes half precision functional unit [52] using the same FPGA (Virtex XCV2000e-6bg560) with [52]. Our design clearly outperforms the reference design in terms of logic area usage and achievable maximum frequency.

Design	Size	Area (Slices)	Max. Freq. (MHz)
Our	8bit x 8	218	73,89
[52]	32 bit x 2	549	61,97
Our	8bit x 16	235	76,36
[52]	64 bit x 2	1023	46,8
Our	8bit x 32	255	72,54
[52]	128 bit x 2	2022	34,73
Our	8bit x 64	256	75,02
[52]	256 bit x 2	3481	18,15

Table 7: Comparison With a Previous Work Using Same FPGAs

An early version of the work can be found in [53]. In the new version, some improvements on total area usage is achieved while we lose some performance in

maximum working frequency. However in most cases, especially the one uses BRAMs, time area product is improved.

3.2 Program and Data Memory

Both program and data memories are realized using RAM inside the FPGA device. Depending on the implementation constraints, distributed or block RAM can be used. However, we prefer to use BRAM since both program and data memories store large amounts of data. Program is stored into the program memory manually before synthesis, and it does not change during the execution. Thus, program memory acts as ROM, instead of RAM. Data memory stores the current values for desired variables. It acts like a register bank. Each core has its own program and data memory. We separated the memories to achieve full parallelism. If the same RAMs are used for each core, then one core should have waited while one core were reading. Moreover control structure of the memory would be more complex. Structure of program and data memory is detailed in the subsequent sections.

3.2.1 Program Memory

As previously indicated program memory is a read-only memory. Program of each core is stored before the synthesis. One program memory utilizes one BRAM but it is addressed into different segments. Each segment is a block program specified for one function. The top controller tells controller to execute code block in BRAM, then controller processes each block line by line. After all the block is finished the controller notifies the top controller. We have a predefined format for micro-instructions in the program memory, which is illustrated in Table 8.

Opcode	Adder/ Subtractor	Op A Core	Op A Index	Op B Core	Op B Index	Result Index	Wait For Core	Notify Core
4 bits	1 bit	1 bit	X bits (5 bits default)	1 bit	X bits (5 bits default)	X bits (5 bits default)	1 bit	1 bit

Table 8: Format of the Micro-Instruction

Definitions related to the micro-instruction are given followings:

- **Opcode**: This is already defined in Table 2. This is the operation code that is to be sent to ACIU.
- **Adder/Subtractor**: Since the same hardware is used for both addition and subtraction, this part defines whether to perform addition or subtraction operation. It will act like adder when this value is '0' and as subtractor, otherwise.
- **Op A Core**: Cores not always operate with operands in their data memory, sometimes they take operands from data memory of other core. This one bit command defines whether the first operand is in data memory of first core or the second. If the value is '0', operand is in the first core, and in the second core, otherwise.
- **Op A Index**: This defines address of the data memory from which the operand is read. Index is a pointer for the address value. Its size is generic but for this application we use it as five bits since we use thirty-two registers.
- **Op B Core**: Same as "Op A Core".
- **Op B Index**: Same as "Op B Index".
- **Result Index**: This defined the register address where the result of the operation is written. Result of each core is written to its own data memory thus, we do not need to specify core number. Again it has a generic size likewise "Op A Index".
- **Wait For Core**: When this value is set to '1', the corresponding FDEU in the controller executes the current micro-instruction, but does not fetch the next micro-instruction. The FDEU waits for a notify core signal (that will come from the other FDEU) before fetching the next instruction. This signal is used to stop execution of the current FDEU (precisely the arithmetic core) and to give the control of its data memory to the other core. This way, we allow one core to access the results in the memory of the other core while the other waits until the memory access operation is completed.

- **Notify Core**: When this value is '1', after the execution of current micro-instruction, one FDEU sends a notify signal to the other FDEU releasing it from the "wait for core" state.

Most attention needs to be paid to the timing of the Notify Core and Wait For Core signals. Firstly, the programmer who develop the micro-codes for the cores to execute, needs to know when to stop and wait to prevent overwrite of the data of one core that will be needed in subsequent clock cycles by the other core. Secondly, we need to know when to notify. In other words, this must not be before the result becomes available so that we must not try to read the data before it is ready. Moreover, we should not make the other core wait longer than necessary. The ideal case for wait and notify chain can be as follows: Data required by one core is generated by the other core and the latter core enters the wait state. Just after that the former core reads desired data from memory of the latter core it notifies the waiting core causing its release from the wait state. To achieve perfect timing the programmer needs to know execution times of micro-instructions.

There can be alternative techniques for inter-core data exchange to simplify the programmer interface. But, a bad alternative is given first to explain the intricate issues in data dependency occurring across the cores. For instance, if a core were to place the other core in "wait for core" state whenever it needs a result from the latter, there would be come complications which are harder to resolve. The fact that it is difficult for a core to know when the other core reaches to the point where the desired result is produced complicates the programming processes.

A successful alternative can be as follows: Before they start executing their micro-codes, cores tell each other which variables they need. Therefore, there are two tasks for two cores: First when these variables are produced, generator core notifies the other core and places itself in "wait" state. Secondly, the core in need of a variable places itself in "wait" state when it reaches to the point in the program where it needs the variable from the other core. When notify signal comes from generator core meaning that the desired variable is available, the latter core leaves the "wait" state, reads the variable, and signals the generator to release it from "wait" state. Naturally, some precautions have to be taken to prevent dead locks, whereby both cores place themselves into wait state since they need each others' variables. Although this method

offers a more user friendly interface for programmer, its hardware implementation is overly complicated. Each core has to monitor the values which are in the desired list of the other core necessitating a check of results of execution of every instruction. Therefore, we prefer to place burden in the programmer side to make the hardware faster and cheaper. Difficulties in developing the codes for the cores are easy to resolve since the programmer can utilize a simple software implementation in high-level language when planning to schedule the micro-codes.

More details about solutions to prevent dead lock are given in Section 3.3. Here we only provide the part relevant to the programmer. The I/O interface of the program memory is given below Figure 11.

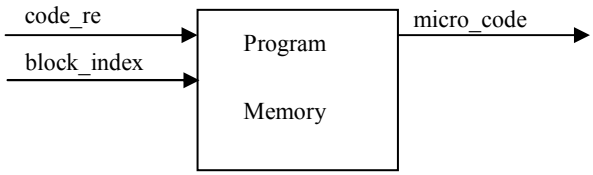

Figure 11: I/O Interface of Program Memory

If “code_re” is active then next micro instruction is given to the controller. During the read of a micro-code for a specific function, “block_index” remains constant.

3.2.2 Data Memory

Data memory is the module where all variables during the program execution are stored. Similar to the program memory, it is divided into abstract segments. Each segment has a constant index. Access to segments is possible using these index numbers. Number of segments in a data memory is generic, thus it can be changed easily. However we use as little segment as possible to efficiently use the whole RAM area. I/O interface of the data memory is given in Figure 12.

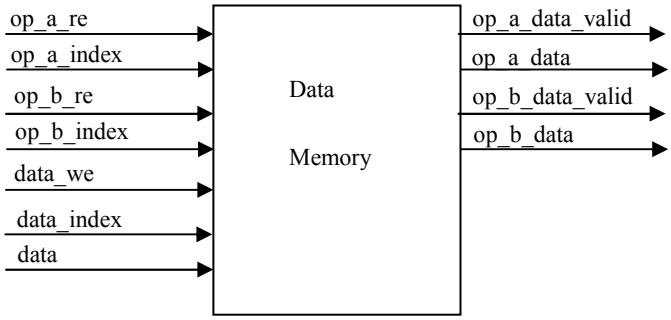

Figure 12: I/O Interface of Data Memory

The data memory is constructed as dual port RAM, which allows synchronous read operations from different addresses. Thanks to this property, we do not need to use two different RAMs to read first operand and second operand in parallel. Data is stored in the specified address (using "data_index" input) when "data_we" is active. Then required operands are read with "op_a_index" and "op_b_index" signals when "op_x_re" is active. It is sufficient to hold "op_x_re" active for one clock cycle, and then data memory makes available the variables at "op_a_data" and "op_b_data" outputs with active valid signals.

3.3 The Controller

The controller is the module which manages the arithmetic cores, program and data memory units. It is responsible of ensuring the correct execution of the micro-code. In fact, to put it simply the controller is akin to a fetch-decode-execute unit. There are two FDEUs inside the controller. To accelerate the communication between FDEUs, they are packed into one module. An FDEU is a state machine and inner states of FDEU are explained below:

- **Initial State**: Initial values of inner variables are set and FDEU waits for a new block of micro-code from the top controller. When top controller gives the index of new block to be executed, this index is used to access the micro-code in the program memory and FDEU proceeds to "check other core" state.

- **Check Other Core State**: This state is designed to prevent a possible collision during memory accesses. In case of two FDEUs try to read operands from the same data memory, a collision may occur in accessing to data

memory. As pointed out earlier, the programmer handles the "wait" and "notify" states correctly to prevent a collision of this type. However, there can still be a collision since the hardware is designed in such a way that a core always reads the others core's data memory with notify signal. Therefore, if an FDEU needs to access the memory of other core two consecutive times for two variables, the latter core is released from the wait state after the first access. Consequently, the latter core continues execution and may access its data memory at the same time with the former core resulting in a collision. To prevent this kind of error from happening, "check other core" state is added before "read operand state". In this state, the core checks the state of the other core. If the other core is in read state then, the core waits for it to finish. If both cores are in this state than we give priority to the first core to avoid dead lock. If the cores are a state other than read state, then it continues to execute normally and passes to "load opcode" state.

- **Load Opcode State**: This state reads the next micro-instruction to execute from the program memory. The number of micro-instruction that have been executed is counted in this state to check if the current program is finished. This state also sends the operation type to the ACIU. After micro-instruction is read and opcode is loaded to ACIU, it proceeds to "read operand" state.

- **Read Operand State**: Depending on micro-instruction, this state reads corresponding registers from the data memory. It waits until all words of the operand are written in ACIU, then "wait for result" state is loaded.

- **Wait For Result State**: As name indicates, here FDEU waits for result of operation is written to the data memory. After that it checks for "wait for core" and "notify core" flags. If one of them is active it branches to the suitable state.

- **Wait For Core State**: In this state FDEU waits for a notify signal from other FDEU or a done signal from inverter controller. As can be remembered inverter has its own controller and program ROM. During the execution of an inversion operation, both cores are placed in "wait for core" state and they both send a notice signal to inverter controller to start. Thus, the inverter

finishes its calculation, it sends a “done” signal which enables both FDEU to wake up. After this, they both proceed to the “check other core” state.

- **Notify Core State**: This state generate the notify signal for the other FDEU. After that, FDEU goes to “check other core” state.

Flow diagram of the state machine is depicted in Figure 13.

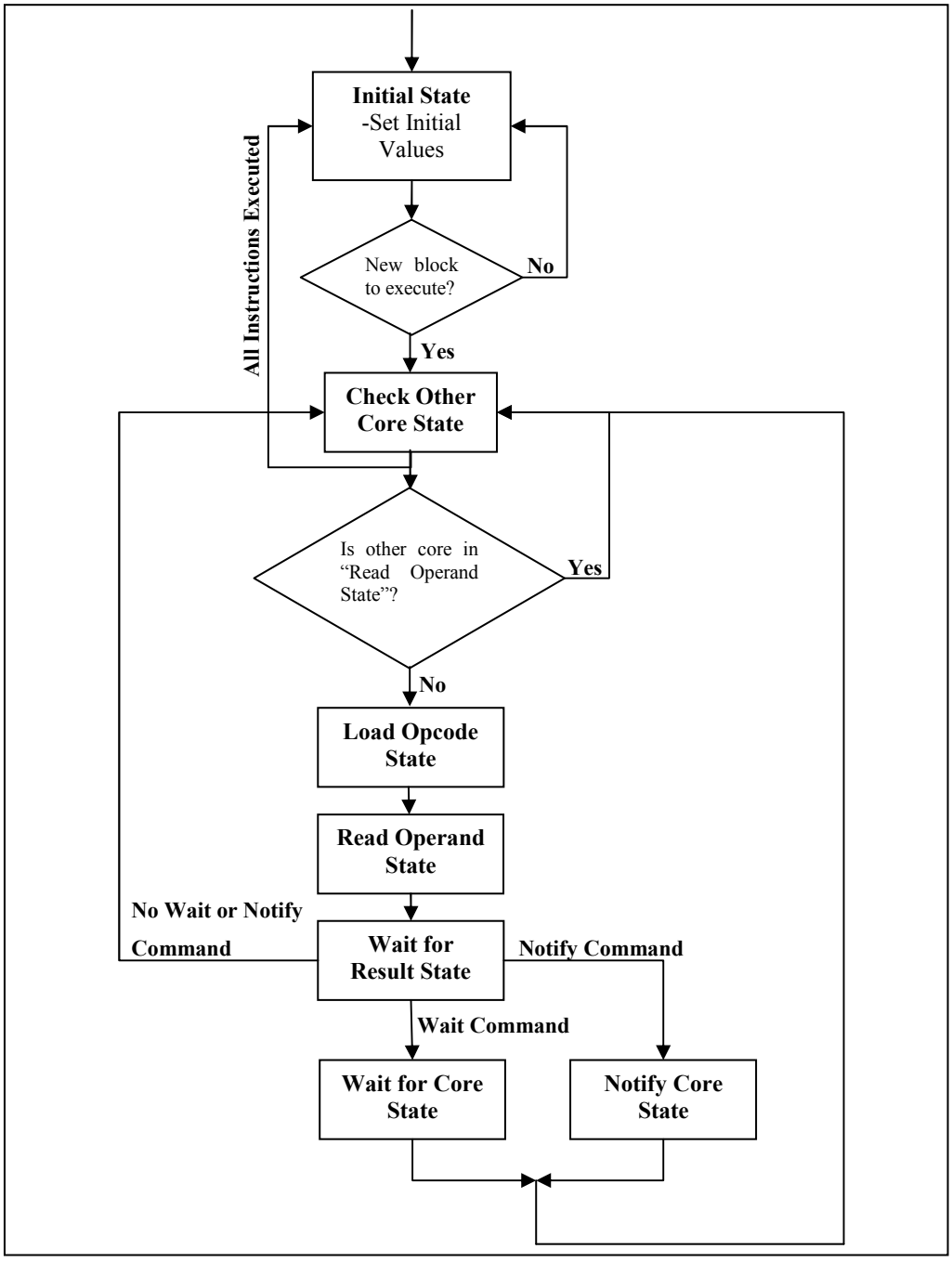

Figure 13: Flow Diagram of State Machine of the Controller

I/O interface of the controller is given in Figure 14.

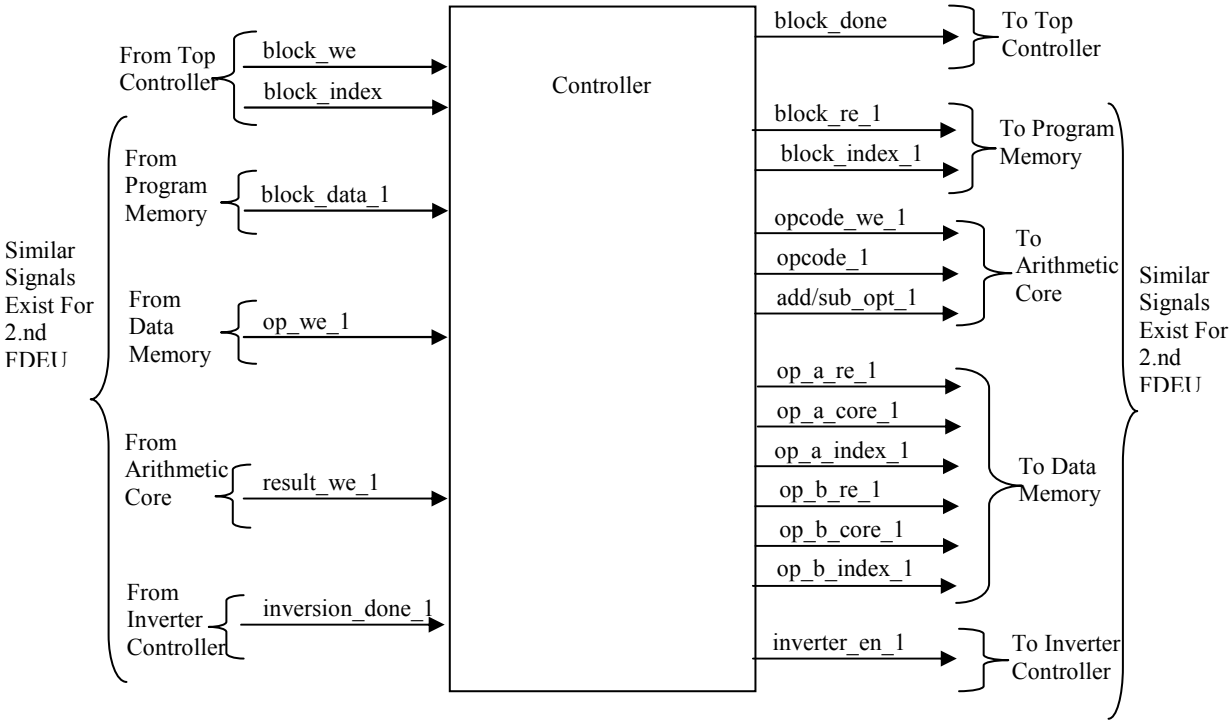

Figure 14: I/O Interface of Controller

In Figure 13, I/O signals of controller are given whereby only signals relating to FDEU1 are shown for sake of simplicity. In fact, all signals with suffix "_1" have versions for FDEU2 as well. Definitions of the I/O ports for the first FDEU is given in Table 9. Similar I/O ports exist for the second FDEU.

I/O Port	Definition
block_index	Defines which block of micro-code to be executed
block_we	When active, block_index is stored to the controller
block_data_1	Micro-instruction read from program memory of 1st. FDEU
op_we_1	Feed-back signal used to indicate if operands are read from data memory
result_we_1	Feed-back signal used to indicate if result of the operation is written to data memory
inversion_done_1	Indicates if inversion is finished
block_done	Indicates that execution of block of micro code is finished
block_index_1	Indicates which block of micro-code is to be read from the 1 st. program memory
block_re_1	When active, micro-instructions, which belongs to "block_index_1", are read in order from the 1 st. program memory
opcode_1	Defines the operation code for 1 st. ACIU
opcode_we_1	When active, "opcode_1" is fed into 1 st. ACIU
add/sub_opt_1	Defines if the adder/subtractor unit in 1 st. ACIU will be used as adder or subtractor
op_a_re_1	Read enable signal (controlled by the 1 st. FDEU) for the first operand.
op_a_core_1	Defines whether the first operand is in the 1 st. data memory or in 2 nd.
op_a_index_1	Defines the address (controlled by the 1 st. FDEU) of the first operand
op_b_re_1	Read enable signal (controlled by the 1 st. FDEU) for the second operand.
op_b_core_1	Defines whether the second operand is in the 1 st. data memory or in 2 nd.
op_b_index_1	Defines the address (controlled by the 1 st. FDEU) of the second operand
inverter_en_1	Signal (controlled by the 1 st. FDEU) used to start inverter

Table 9: I/O Port Definitions for the First FDEU

3.4 The Top Controller

Top controller is the only part of the design specific for a given application. Other parts of the processor may be used in other applications directly. But the top controller has to be modified to implement other cryptographic or other pairing operations. The

top controller informs the controller to execute a block of micro-code. A cryptographic application is first divided into computational segments for each of which a micro-code block is developed. These micro-code blocks are placed in the program memory. Therefore, the top controller should be designed in such a way that these micro-code blocks are given to the FDEUs in proper order. Consequently, the top controller should be re-designed for every application. We explain the design of the top controller for Algorithm 1.

We construct two blocks of micro-code inside the "for" loop in Algorithm 1 (steps 1-8). The first block represents the operations to be performed when $r_i = 1$ (steps 2 and 3 of Algorithm 1) and the other represents the operations when $r_i = 0$ (steps 4 and 5 of Algorithm 1). The top controller controls the "for" loop and the final exponentiation operation (step 9 of Algorithm 1). Variables related to the "for" loop (r) and final exponentiation ($p^k - 1/r$) are stored in the inner RAMs of top controller. Then the top controller shifts *r* by one bit (step 1 of Algorithm 1) and informs the controller module with the micro-code block to execute. When "done" signal is received from the controller, the top controller shifts *r* again and sends next block of micro-code. After the for loop terminates, it instructs the controller to execute block of code related to the final exponentiation. Finally, the top controller raises a finish flag indicating that the final result is ready. Inner abstraction and I/O interface of top controller is depicted in Figure 14.

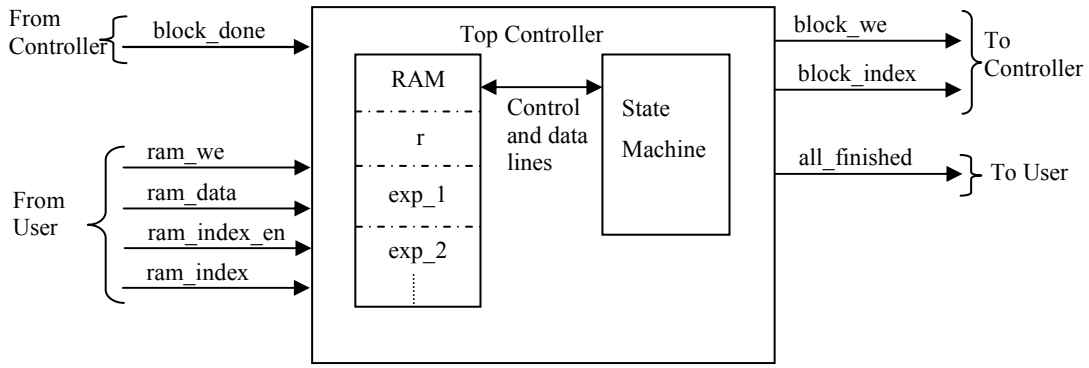

Figure 15: I/O Interface and inner abstraction of top controller

Only one block RAM used to store variables of the top controller. Each variable is addressed with a separate index. Before starting of execution, variables needed for the top controller have to be stored. Using "ram_index_en" and "ram_index" signals, initial variables are written in the RAM, whereby "ram_index" are set to the address of the

location of the corresponding variable in the RAM. During the write operations, data is applied to the “ram_data” with “ram_we” signal being set active. Another module or a separate circuit using the pairing coprocessor can perform these operations. After the actual operation (e.g. Tate pairing) is completed, “all_finished” flag is set active. One block of code actually contains the parts for both cores which are marked by the programmer to assign them to the cores. The entire program installed on the processor is given block by block in the Appendix.

3.5 Debugging of the Hardware

For test purposes, Tate pairing algorithm is implemented using MIRACL C++ crypto library. We added print-outs for the intermediate results, such as the result of line evaluation function, output of each iteration of the Miller loop and intermediate and final steps of the final exponentiation. After all design and HDL coding are finished, we run behavioral simulation with the input values used in MIRACL implementation. We compared our results with MIRACL outputs to fix the bugs in HDL implementation. However, behavioral simulation takes around 30 minutes to complete and after each fix it takes too long to complete the simulation for the new implementation. To remedy the slowness in hardware debugging, we developed an emulator for the hardware in Java language. We run the Java program and observed that its output and all intermediate results match with those of the hardware simulation. Then, we continued testing with the Java emulator and MIRACL to match the results of both of them. By this way we were able to find the bugs in a shorter time compared to previous method. Then we transferred the changes we made in the Java emulator to the hardware implementation. Finally, we run behavioral simulation and MIRACL and we observed that all the results are matched.

4 Conclusion and Comparison

We design a general purpose pairing processor for FPGAs. Our design is a parametric, very flexible and a very compact implementation. It can even fit into one (Spartan-3s400) of the very old fashioned and small FPGAs (see Table 9). Also it performs higher working frequency compared to the similar work [54].

During improvement process, we made behavioral unit tests for each sub-module of the design and for overall design using ISIM and ModelSim simulation environment. After behavioral simulation we tried our design on real hardware (Xilinx ML402 Evaluation Board) using 100MHz clock source. We see that our design works correctly. Our design can be easily modified to work on different types of pairing operations. Among many pairing operation types we chose to implement Tate pairing [4] using the parameters given in the MIRACL crypto library. We use the elliptic curve:

$$y^2 = x^3 - 3x + B$$

$B = 364450518177934192404424328677091614072588218039367457522428451192010880$

2552.We use the modulus, $M = 330834540866291994040950336878685123996049234435$

$076633578656844659271970754 53$. We constructed quadratic field $F_{q^2} - \beta$ using $\beta = -2$. We choose the r variable in the Algorithm 1 as $r = 25803063399904061661127$

$976311108304689363428717269$. The generator point P in Algorithm 1 whose coordinates are on F_q is chosen as follows.

$P_x = 640227261954506466835518614091388818629251204730213225696365875116607958$

2595

$P_y = 282547540334082505540596921915739384162084332653661886191994405437458857967792$

The point Q in Algorithm 1, which is on F_{q^4}, is chosen as follows:

$Q_{x_{0,1}} = 202468240713654861785554933261760322244078402596801989724654122093464266322,$

$Q_{x_{0,0}} = 0,$

$Q_{x_{1,0}} = 0,$

$Q_{x_{1,1}} = 0,$

$Q_{y_{1,0}} = 139159510539237898486697191650467806305180186022574981390954112029811036673335,$

$Q_{y_{1,1}} = 879873655279966654546263390920582730997142969899536448244002331049178 1584610,$

$Q_{y_{0,0}} = 0,$

$Q_{y_{0,1}} = 0.$

By using the parameters above, after running placement-and-routing (PAR) for the target device, we obtain the following results for a prime q of 255 bits, where r is 160 bits prime integer and 80 bit of security is intended. The word size is chosen as 15 bits to utilize the hardwired multipliers. Total running time of the Tate pairing is found to be 28.65ms, 54.92% of which is spent on the Miller loop, using a clock frequency of 132.49 MHz. The design consumes 4829 registers, 7583 LUTs and 4 BRAMs. Our results can be seen in Table 9.

m	WL	REG	Slice	LUT	MF	T(ms)	Security	FPGA
256	4	1557	2394	4424	78	?	80	Spartan3s400
255	15	4293	-	6198	132.49	28.65	80	Spartan 6
255	15	4321	6071	10362	128.68	29.49	80	Virtex 4

Table 10: PAR Results for Co-processor Implementing Tate Pairing

We implemented design for both Spartan-6 and Virtex-4 for comparison purposes as shown in Table 9. In a recent PhD thesis published in June 2011 [54], the author proposes a similar architecture. The author provides implementation results for 128 bits security on Virtex 4 (xc4vlx200). It achieves a maximum clock frequency of 50 MHz and 35.3 ms completion time. Although it performs higher security level, it is clear that our design outperforms in terms of maximum achievable working frequency and logic area usage. It is important to note that, since our design is word based, logic area usage of our design does not increase significantly with increase of modulus size, instead total completion time of our design increases with the modulus bit sizes. Thus we can still say that for improved security level our logic area usage will increase slightly and our achievable maximum frequency can decrease slightly. Anyway, it will be better in terms of MF and logic area usage compared to [54] even for 128 bit security. Also we give comparison with an ASIC design satisfying 128 bits security [55]. Even if this design is ASIC, our design outperforms clearly in terms of logic usage.

Design	**WL**	**REG**	**Slice**	**LUT**	**MF**	**T(ms)**	**Security**	**FPGA**
Our(255)	15	4829	-	7583	132.49	28.65	80	Spartan 6
Our(255)	15	4856	6551	11241	128.68	29.49	80	Virtex 4
[54]	-	27k	52k	101k	50	35.3	128	Virtex 4
[55]	-	-	97k	-	338	34.4	128	ASIC-130nm

Table 11: Comparison Results

References

[1] A. J. Menezes, T. Okamoto, and S. A. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field," *IEEE Trans. Info. Theory, 39:1639–1646,* 1993.

[2] D. Boneh and M. Franklin, "Identity-based encryption from the Weil pairing", *SIAM Journal of Computing, 32(3):586-615*, 2003.

[3] J. Fan, F. Vercauteren, and I. Verbauwhede, "Efficient Hardware Implementation of F*p*-arithmetic for Pairing-Friendly Curves", *IEEE Transaction on Computers*, 2011.

[4] A. Devegili, M. Scott, and R. Dahab, "Implementing Cryptographic Pairings over Barreto-Naehrig Curves", *Pairing 2007, volume 4575 of Lecture Notes in Computer Science, pages 197–207. Springer*,2007.

[5] Xilinx Company, "Spartan-6 family overview", http://www.xilinx.com/support/documentation /ds160.pdf, 3 March 2010.

[6] Xilinx Company, "Spartan-6 fpga configurable logic block user guide",http://www.xilinx.com/ support/documentation/user guides/ug384.pdf, 23 February 2010.

[7] Digi-Key Corporation, http://search.digikey.com/scripts/DkSearch/dksus.dll?Cat=2556262&k= XC6SLX45T, 1 August 2011.

[8] Xilinx Company, "IP Processor Block Ram (BRAM) Block",http://www.xilinx.com/support /documentation/ip_documentation/bram_block.pdf, 1 August 2011.

[9] Xilinx Company, "Spartan-6 FPGA DSP48A1 Slice User Guide",http://www.xilinx.com/support /documentation/user_guides/ug389.pdf, 1 August 2011.

[10] R. Sakai, K. Ohgishi and M. Kasahara, "Cryptosystems based on pairing", *2000 Symposium on Cryptography and Information Security-SCIS 2000, pages 26-28,*Okinawa, Japan, Jan.2000.

[11] A. Joux, "A one round protocol for tripartite DiffieCHellman", *ANTS-4: Algorithmic Number Theory. Springer-Verlag, volume 1838 of Lecture Notes in Computer Science, pp. 385-394. Springer-Verlag*, 2000.

[12] Website of Department of Computer Science City University of Hong Kong, http://www.cs.cityu.edu.hk/~ecc/ home.htm, 1 August 2011.

[13] S. Galbraith, K. Harrison, and D. Soldera, "Implementing the Tate pairing", *In Algorithm Number Theory Symposium - ANTS V, volume 2369 of Lecture Notes in Computer Science, pp. 324-337. Springer-Verlag*, 2002.

[14] R. Granger, F. Hess, R. Oyono, N. Th´eriault, and F. Vercauteren, “Ate pairing on hyperelliptic curves”, *Advances in Cryptology - EUROCRYPT 2007, volume 4515 of Lecture Notes in Computer Science, pp. 430–447. Springer-Verlag*, 2007.

[15] N. P. Smart, “An identity based authenticated key agreement protocol based on the weil pairing”, *Electronics Letters, 38:630–632*, 2002.

[16] M. Scott and P. S. L. M. Barreto, “Compressed pairings”, In *Crypto 2004*, 2004.

[17] A. Miyaji, M. Nakabayashi, and S. Takano, “New explicit conditions of elliptic curve traces for FR-reduction”, *IEICE Transactions on Fundamentals, E84-A(5):1234 – 1243*, 2001.

[18] D. F. Aranha, J. L´opez, and D. Hankerson, “High-speed parallel software implementation of the η*T* pairing”, *CT-RSA 2010, LNCS 5985, pp. 89–105. Springer*, 2010.

[19] P. Grabher, J. Großschadl, and D. Page, “On software parallel implementation of cryptographic pairings”, *SAC 2008. LNCS 5381, pp. 35-50*, 2008.

[20] J. L. Beuchat, J. E. G. Diaz, S. Mitsunari, E. Okamoto, F. R. Henriquez, and T. Teruya, “High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves”, *Pairing 2010, LNCS 6487, pp. 21–39,* 2010.

[21] Xilinx Company, “Spartan-3 FPGA Family Data Sheet”, http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf, 1 August 2011.

[22] A. Karatsuba and Y. Ofman, "Multiplication of Many-Digital Numbers by Automatic Computers", *Proceedings of the USSR Academy of Sciences 145: 293–294*, 1962.

[23] M. Scott, “Computing the Tate pairing”, *CT-RSA, volume 3376 of Lecture Notes in Computer Science, pages 293-304. Springer-Verlag*, 2005.

[24] G. M. Bertoni , L. Chen, P. Fragneto, K. A. Harrison, G. Pelosi, “Computing tate pairing on smartcards”, http://www.st.com/stonline/products/families/smartcard/ches2005v4.pdf, 2005.

[25] S. Chatterjee, P. Sarkar, and R. Barua, “Efficient computation of Tate pairing in projective coordinate over general characteristic fields”, *ICISC 2004, LNCS 3506, pp. 168-181*, 2005.

[26] J. H. Silverman, “The arithmetic of elliptic curves”, *Springer GTM 106*, 1986.

[27] V. S. Miller, “The Weil pairing, and its efficient calculation”, *Journal of Cryptology, 17(4):235–261,* September 2004.

[28] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott, “Efficient algorithms for pairing-based cryptosystems”, In *CRYPTO 2002*, number 2442 in *Lecture Notes in Computer Science, pp. 354–369, Springer-Verlag,* Berlin Heidelberg, 2002.

[29] N. E. Mrabet, S. Ionica and N. Guillermin, “Pairing computation at 192 bits level security”, http://www.ai.univ-paris8.fr/~elmrabet/Article/articlek15v14.pdf, 2011.

[30] Recommendations for Key Management, Special Publication 800-57 Part 1, 2007.

[31] L. Hitt, "On the minimal embedding field." In *Pairing-Based Cryptography — Pairing 2007, Springer LNCS 4575, pp. 294–301*, 2007.

[32] D. Freeman, M. Scott, and E. Teske, "A Taxonomy of Pairing-Friendly Elliptic Curves", *Journal of Cryptology, 23(2):224–280*, 2010.

[33] IEEE P1363. Standard Specifications for Public Key Cryptography. IEEE, 2000.

[34] C. H. Lim, H. S. Hwang, "Fast Implementation of Elliptic Curve Arithmetic in GF (p^m)", *Proc. PKC, LNCS 1751*, 2000.

[35] A. Weimerskirch, and C. Paar, "Generalizations of the Karatsuba Algorithm for Efficient Implementations", http://www.crypto.ruhr−uni−bochum.de/imperia/md/content/texte/kaweb.pdf, 2003.

[36] M. Scott, "On the Efficient Implementation of Pairing-Based Protocols", *Cryptology ePrint Archive, Report 2011/334*, 2011.

[37] J. C. Bajard and N. E. Mrabet, "Pairing in cryptography: an arithmetic point of view", http://www.ai.univ-paris8.fr/~elmrabet/Presentation/SPIE07_talk_ELMRABET.pdf, 2007.

[38] Y. Nogami, M. Akane, Y. Sakemi, Y. Morikawa, "Efficient Pairings on Twisted Elliptic Curve", *ICCIT '08. Third International Conference on*, 2008.

[39] D. M. Gordon, "A survey of fast exponentiation methods", *Journal of Algorithms 27, 129–146. ISSN 0196-6774*, 1998.

[40] P. L. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation, vol. 44, pp.519-521,* 1985.

[41] E. Oksuzoglu, E. Savas, "Parametric, Secure and Compact Implementation of RSA on FPGA", *Reconfigurable Computing and FPGAs*, 2008.

[42] C. McIvor, M. Mcloone, J. N. McCanny, A. Daly, W. Marnane, "Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures", *37th Annual Asilomar Conference on Signals, Systems and Computers,* California , 2003.

[43] NIST, "Digital Signature Stantard (DSS)", *FIPS PUB186-2*, 2000.

[44] J. J. Quisquater and C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystem", *Electronics Letters, vol. 18, pp. 905–907*, 1982.

[45] D. Hankerson, A. J. Menezes, Scott Vanstone, *Guide to Elliptic Curve Crptography*, Springer, Norwell, 2004.

[46] A. J. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, Norwell, MA, USA, 1994.

[47] N. Koblitz, “Elliptic curve cryptosystems”, *Mathematics of Computation, vol. 48, pp. 203–209*, January 1987.

[48] T. Kobayashi, “Fast modular inversion algorithm to match any operation unit”, *IEICE TRANS. FUNDAMENTALS, vol. 82, no. 5, pp. 733–740*, 1999.

[49] B. S. Kaliski, “The montgomery inverse and its applications”, IEEE *Trans. Comput., vol. 44, no. 8, pp. 1064–1065*, 1995.

[50] D. E. Knuth, *Art of Computer Programming*, Volume 2: Seminumerical Algorithms, Addison-Wesley, 1998.

[51] E. Savas and C. K. Koc, “The montgomery modular inverse revisited”, *IEEE Trans. Computers, vol. 49, no. 7, pp. 763–766*, 2000.

[52] E. Popovici, A. Daly, W. Marnane, “Fast modular inversion in the montgomery domain on reconfigurable logic”, *Proc. of ISSC, pp. 362–367*, 2003.

[53] E. Murat, S. Kardaş, E. Savaş, “Scalable and Efficient FPGA implementation of Montgomery Inversion “, *Workshop on Lightweight Security & Privacy: Devices, Protocols, and Applications*, 2011.

[54] S. Ghosh, D. Mukhopadhyay, and D.R. Chowdhury, “High Speed Flexible Pairing Cryptoprocessor on FPGA Platform”, In *Pairing 2010, volume 6487 of Lecture Notes in Computer Science, pp. 450–466*, 2010.

[55] D. Kammler, D. Zhang, P. Schwabe, H. Scharwaechter, M. Langenberg,D. Auras, G. Ascheid, R. Leupers, R. Mathar, and H. Meyr, “Designing an ASIP for Cryptographic Pairings over Barreto-Naehrig Curves”, In CHES *2009, volume 5747 of Lecture Notes in Computer Science, pp. 254–271. Springer*, 2009.

Appendix

Computing of $f \leftarrow f^2 * l_{T,T}(Q)\ and\ T \leftarrow T + T$

F = F0 + F1.I = (f0,0 + i.f0,1) + (f1,0 + i.f1,1).I

	ACORE 0			ACORE 1	
Micro Code	**Operation**	**Explanation**	**Micro Code**	**Operation**	**Explanation**
t0 = f0,0			t0' = Xa		
t1 = f0,1			t1' = Ya		
t2 = f1,0			t2' = Za		
t3 = f1,1			t3' = xq0,0		Real
t4 = Xb			t4' = xq0,1		Imaginary
t5 = Yb			t5' = yq1,0		Real
t6 = 0			t6' = yq1,1		Imaginary
t30 = 1			t7' = 0		
t31 = β			t28' = a		
			t29' = β		
			t30' = Fr	Frobenious	Real
			t31' = i.Fr	Frobenious	Imaginary
	F^2	**13M, 28A**		**Lt,t(Q)**	**17M, 5A**
t7 = t0.t0	T1 = F0.F0		t8' = 3.t0'.t0'		
t8 = t1.t1			t9' = (t2')^3		Za^3
t9 = β.t8			t10' = t9'.t2'		
t10 = t7 + t9		T1 Real	t10' = t10'.a		
t9 = t7 + t8			t8'= t8' + t10'		λa,a
t7 = t0 + t1			t10' = 2.t1'.t2'		Zc, Do not overwrite
t8 = t0 + t1			t11'= t5'.t9'		
t7 = t7.t8			t11' = t11'.t10'		Lt,t(Q)1,0
t11 = t7 – t9		T1 Imaginary	t11' = t7' – t11'		
t7 = t2.t2	T2 = F1.F1		t12' = t6'.t9'		
t8 = t3.t3			t12' = t12'.t10'		Lt,t(Q)1,1
t9 = β.t8			t12' = t7'- t12'		
t12 = t7 + t9		T2 Real	t13' = t0'.t2'		
t9 = t7 + t8			t13' = t8'.t13'		λa,a.Xa.Za
t7 = t2 + t3			t14' = t3'.t9'		Wait For Core 0
t8 = t2 + t3					
t7 = t7.t8					
t13 = t7 –t9		T2 Imaginary			
t14 = β.t13	T3 = γ.T2 = i(t2,0 + i.t2,1)=β.t2,1 + i.t2,0	T3 Real			
t15 = t6 + t12		T3 Imaginary			
t16= t10+t14	C0 = T1 + T3	C0 Real			
t17= t11+t15		C0 Imaginary			

t14= t10+t12	T3 = T1 + T2	T3 Real			
t15= t11+t13		T3 Imaginary			
t10= t0 + t2	T1 = F0 + F1	T1 Real			
t11= t1 + t3		T1 Imaginary			
t12= t0 + t2	T2 = F0 + F1	T2 Real			
t13= t1 + t3		T2 Imaginary			
t7 = t10.t12	T4 = T1.T2				
t8 = t11.t13					
t9 = β.t8					
t18= t7 + t9		T4 Real			
t9 = t7 + t8					
t7 = t10 + t11					
t8 = t12 + t13					
t7 = t7.t8					
t19 = t7 – t9		T4 Imaginary			
t0 = t16 + t6	C1 = T4-T3	C0 0,0			
t1 = t17 + t6		C0 0,1			
t2 = t18 – t14		C1 1,0			
t3 = t19 – t15		C1 1,1			
	Doubling	**5M, 6A**			
t10 = t6 + t0’		Xa			
t11 = t6 + t1’		Ya			
t7 = t6 + t8’		λa,a / Notify Core 1			
t11 = 2.t11.t11			t14’ = t14’.t8’		Real
t10 = 2.t10.t11			t15’ = t4’.t9’		
t8 = t7.t7			t15’ = t15’.t8’		Imaginary / Lt,t(Q)0,1
t9 = 2.t10			t16’ = t1’.t10’		
t8 = t8 – t9		Xc	t14’ = t13’ – t14’		
t10 = t10 – t8			t14’ = t14’ – t16’		
t10 = t10.t7			t14’ = t7’ –t14’		Lt,t(Q)0,0 / Wait For Core 0
t11 = 2.t11.t11					
t10 = t10 – t11		Yc			
t20 = t6 + t14’		Lt,t(Q)0,0			
t21 = t6 + t15’		Lt,t(Q)0,1 / Notify Core1			
t6 = t6 + t6	Dummy	Wait For Core 1	t17’ = t7’ + t2		F 1,0
			t18’ = t7’ + t3		F 1,1
			t0’ = t8 + t7’		
			t1’ = t10 + t7’		
			t2’ = t10’ + t7’		
			t7’ = t7’ + t7’	Dummy	Notify Core 0
	f^2.Lt,t(Q)			**f^2.Lt,t(Q)**	
t7 = t0.t20	T1 = A0.B0	A0 -> f, B0->Lt,t(Q)	t8’ = t17’.t11’	T2 = A1.B1	A1 -> f, B1->Lt,t(Q)
t8 = t1.t21			t9’ = t18’.t12’		
t9 = β.t8			t10’ = β.t9’		
t10 = t7 + t9		T1 Real	t19’ = t8’ + t10’		T2 Real
t9 = t7 + t8			t10’ = t8’ + t9’		
t7 = t0 + t1			t8’ = t17’ + t18’		

t8 = t20 + t21			t9’ = t11’ + t12’		
t7 = t7.t8			t8 ‘ = t8’.t9’		
t11 = t7 – t9		T1 Imaginary / Wait For Core 1	t20’ = t8’ – t10’		T2 Imaginary
			t21’ = β.t20’	T3 = γ.T2 = i(t2,0 + i.t2,1)=β.t2,1 + i.t2,0	T3 Real
			t22’ = t7’ + t19’		T3 Imaginary / Notify Core 0
t6 = t6 + t6	Dummy		t7’ = t7’ + t7’	Dummy	Wait For Core 0
t6 = t6 + t6	Dummy				
t16 = t10 + t21’	C0 = T1 + T3	C0 Real			
t17 = t11 + t22’		C0 Imaginary			
t14 = t10 + T19’	T3 = T1 + T2	T3 Real			
t15 = t11 + t20’		T3 Imaginary / Notify Core 1			
t10 = t0 + t2	T1 = A0 + A1	T1 Real	t19’ = t11’ + t14’	T2 = B0 + B1	T2 Real
t11 = t1 + t3		T1 Imaginary	t20’ = t12’ + t15’		T2 Imaginary / Wait For Core 0
t7 = t10.t19’	T4 = T1.T2				
t8 = t11.t20’		Notify Core 1			
t9 = β.t8			t21’ = t19’+t20’		Wait For Core 0
t18 = t7 + t9		T4 Real			
t9 = t7 + t8		Notify Core 1			
t7 = t10 + t11			t7’ = t7’ + t7’	Dummy	To provide enough latency for Acore0
t7 = t7.t21’				DONE	
t19 = t7 – t9		T4 Imaginary			
t0 = t16 + t6	C1 = T4-T3	C0 0,0			
t1 = t17 + t6		C0 0,1			
t2 = t18 – t14		C1 1,0			
t3 = t19 – t15		C1 1,1			
	DONE				

Computing of $f \leftarrow f * l_{T,P}(Q)\ and\ T \leftarrow P + T$

	ACORE 0			ACORE 1	
Micro Code	**Operation**	**Explanation**	**Micro Code**	**Operation**	**Explanation**
t0 = f0,0			t0’ = Xa		
t1 = f0,1			t1’ = Ya		
t2 = f1,0			t2’ = Za		
t3 = f1,1			t3’ = xq0,0		Real
t4 = Xb			t4’ = xq0,1		Imaginary
t5 = Yb			t5’ = yq1,0		Real

t6 = 0			t6’ = yq1,1		Imaginary
t30 = 1			t7’ = 0		
t31 = β			t28’ = a		
			t29’ = β		
			t30’ = Fr	Frobenious	Real
			t31’ = i.Fr	Frobenious	Imaginary
	T+P			**Lt,p(Q)**	
t6 = t6 + t6	Dummy		t7’ = t7’ + t7’	Dummy	Wait For Core 0
t7 = t6 + t0’		Xa			
t8 = t6 + t1’		Ya			
t9 = t6 + t2’		Za / Notify Core 1			
t10 = t9.t9			t8’ = t0’.t2’		Wait For Core 0
t11 = t10.t9		Notify Core 1			
t12 = t11.t5			t9’ = t7’ + t11		Za^3 / Wait For Core 0
t12 = t12-t8		λa,b / Notify Core 1			
t10 = t10.t4			t10’ = t7’ + t12		λa,b
t10 = t10 – t7			t11’ = t8’.t10’		
t13 = t9.t10		Zc / Wait For Core 1	t12’ = t9’.t10’		
			t7’ = t7’ + t7’	Dummy	
			t13’ = t7’ + t13		Zc / Notify Core 0
t14 = t10.t10			t14’ = t3’.t12’		Real
t15 = t14.t10			t15’ = t4’.t12’		Imaginary
t14 = t14.t7			t16’ = t1’.t13’		
t10 = t12.t12			t14’ = t11’ – t14’		
t16 = t15 +t14			t14’ = t14’ – t16’		Lt,p(Q)0,0
t16 = t16 + t14			t15’ = t7’ – t15’		Lt,p(Q)0,1
t10 = t10 –t16		Xc	t9’ = t9’.t13’		
t14 = t14 – t10			t11’ = t5’.t9’		Lt,p(Q)1,0
t12 = t12.t14			t12’ = t6’.t9’		Lt,p(Q)1,1 / Wait For Core 0
t14 = t15.t8					
t12 = t12 – t14		Yc			
t20 = t6 + t14’		Lt,t(Q)0,0			
t21 = t6 + t15’		Lt,t(Q)0,1 / Notify Core1			
t6 = t6 + t6	Dummy	Wait For Core 1	t17’ = t7’ + t2		F 1,0
			t18’ = t7’ + t3		F 1,1
			t0’ = t7’ + t10		
			t2’ = t7’ + t13’		
			t1’ = t7’ + t12		Notify Core 0
	f.Lt,p(Q)			**f.Lt,p(Q)**	

t7 = t0.t20	T1 = A0.B0	A0 -> f, B0->Lt,t(Q)	t8' = t17'.t11'	T2 = A1.B1	A1 -> f, B1->Lt,t(Q)
t8 = t1.t21			t9' = t18'.t12'		
t9 = β.t8			t10' = β.t9'		
t10 = t7 + t9		T1 Real	t19' = t8' + t10'		T2 Real
t9 = t7 + t8			t10' = t8' + t9'		
t7 = t0 + t1			t8' = t17' + t18'		
t8 = t20 + t21			t9' = t11' + t12'		
t7 = t7.t8			t8 ' = t8'.t9'		
t11 = t7 – t9		T1 Imaginary / Wait For Core 1	t20' = t8' – t10'		T2 Imaginary
			t21' = β.t20'	T3 = γ.T2 = i(t2,0+i.t2,1)=β.t2,1 + i.t2,0	T3 Real
			t22' = t7' + t19'		T3 Imaginary / Notify Core 0
t6 = t6 + t6	Dummy		t7' = t7' + t7'	Dummy	Wait For Core 0
t6 = t6 + t6	Dummy				
t16 = t10 + t21'	C0 = T1 + T3	C0 Real			
t17 = t11 + t22'		C0 Imaginary			
t14 = t10 + T19'	T3 = T1 + T2	T3 Real			
t15 = t11 + t20'		T3 Imaginary / Notify Core 1			
t10 = t0 + t2	T1 = A0 + A1	T1 Real	t19' = t11' + t14'	T2 = B0 + B1	T2 Real
t11 = t1 + t3		T1 Imaginary	t20' = t12' + t15'		T2 Imaginary / Wait For Core 0
t7 = t10.t19'	T4 = T1.T2				
t8 = t11.t20'		Notify Core 1			
t9 = β.t8			t21' = t19'+t20'		Wait For Core 0
t18 = t7 + t9		T4 Real			
t9 = t7 + t8		Notify Core 1			
t7 = t10 + t11				DONE	
t7 = t7.t21'					
t19 = t7 – t9		T4 Imaginary			
t0 = t16 + t6	C1 = T4-T3	C0 0,0			
t1 = t17 + t6		C0 0,1			
t2 = t18 – t14		C1 1,0			
t3 = t19 – t15		C1 1,1			
	DONE				

Final exponentiation part 1

	ACORE 0			ACORE 1	
Micro Code	**Operation**	**Explanation**	**Micro Code**	**Operation**	**Explanation**

t0 = f0,0		loop output	t0’ = Xa		
t1 = f0,1		loop output	t1’ = Ya		
t2 = f1,0		loop output	t2’ = Za		
t3 = f1,1		loop output	t3’ = xq0,0		Real
t4 = Xb			t4’ = xq0,1		Imaginary
t5 = Yb			t5’ = yq1,0		Real
t6 = 0			t6’ = yq1,1		Imaginary
t30 = 1			t7’ = 0		
t31 = β			t28’ = a		
			t29’ = β		
			t30’ = Fr	Frobenious	Real
			t31’ = i.Fr	Frobenious	Imaginary
	B = (F0 + IF1)^-1				
t7 = t2.t2	T1 = A1.A1		t7’ = t7’ + t7’	Dummy	Wait For Core 0
t8 = t3.t3					
t9 = β.t8					
t10 = t7 + t9		T1 Real			
t9 = t7 + t8					
t7 = t2 + t3					
t8 = t2 + t3					
t7 = t7.t8					
t11 = t7 – t9		T1 Imaginary			
t12 = β.t11	T2 = γ.T1 = (β.t1,1 + i.t1,0)	T2 Real			
t13 = t6 + t10		T2 Imaginary			
t7 = t0.t0	T1 = A0.A0				
t8 = t1.t1					
t9 = β.t8					
t10 = t7 + t9		T1 Real			
t9 = t7 + t8					
t7 = t0 + t1					
t8 = t0 + t1					
t7 = t7.t8					
t11 = t7 – t9		T1 Imaginary			
t14 = t10 – t12	T3 = T1-T2	T3 Real			
t15 = t11 – t13		T3 Imaginary			
t7 = t15.t15	T4 = T3^-1				
t8 = β.t7					
t7 = t14.t14					
t9 = t7 – t8		Wait For Core 1			
t9 = t9^-1	This step is executed by Inverter Controller, it does not exist in program memory of the controller				
t18 = t14.t9		T4 Real	t7’ = t7’ + t7’	Dummy	Wait For

					Core 0
t19 = t15.t9					
t19 = t6 – t19		T4 Imaginary			
t7 = t0.t18	B0 = A0.T4				
t8 = t1.t19					
t9 = β.t8					
t20 = t7 + t9		B0,0			
t9 = t7 + t8					
t7 = t0 + t1					
t8 = t18 + t19					
t7 = t7.t8					
t21 = t7 – t9		B0,1			
t10 = t6 – t2	T1 = -A1	T1 Real			
t11 = t6 – t3		T1 Imaginary			
t7 = t10.t18	B1 = T1.T4				
t8 = t11.t19					
t9 = β.t8					
t22 = t7 + t9		B1,0			
t9 = t7 + t8					
t7 = t10 + t11					
t8 = t18 + t19					
t7 = t7.t8					
t23 = t7 – t9		B1,1			
	B.F **F=(F0 +I(-F1)**				
t2 = t6 – t2	Generate (-F1)				
t3 = t6 – t3	Generate (-F1)				
t7 = t0.t20	T1 = F0.B0				
t8 = t1.t21					
t9 = β.t8					
t10 = t7 + t9		T1 Real			
t9 = t7 + t8					
t7 = t0 + t1					
t8 = t20 + t21					
t7 = t7.t8					
t11 = t7 – t9		T1 Imaginary			
t7 = t2.t22	T2 = F1.B1				
t8 = t3.t23					
t9 = β.t8					
t12 = t7 + t9		T2 Real			
t9 = t7 + t8					
t7 = t2 + t3					
t8 = t22 + t23					
t7 = t7.t8					
t13 = t7 – t9		T2 Imaginary			
t14 = β.t13	T3 = γ.T2 = i(t2,0+i.t2,1)= β.t2,1 + i.t2,0	T3 Real			
t15 = t6 + t12		T3 Imaginary			
t16= t10+t14	C0 = T1 + T3	C0 Real			
t17= t11+t15		C0 Imaginary			
t14= t10+t12	T3 = T1 + T2	T3 Real			
t15= t11+t13		T3 Imaginary			
t10= t0 + t2	T1 = F0 + F1	T1 Real			
t11= t1 + t3		T1 Imaginary			
t12= t20 + t22	T2 = B0 + B1	T2 Real			
t13= t21 + t23		T2 Imaginary			

t7 = t10.t12	T4 = T1.T2				
t8 = t11.t13					
t9 = β.t8					
t18= t7 + t9		T4 Real			
t9 = t7 + t8					
t7 = t10 + t11					
t8 = t12 + t13					
t7 = t7.t8					
t19 = t7 – t9		T4 Imaginary			
t0 = t16 + t6	C1 = T4-T3	C0 0,0		Prepare Exponentiation s^k1	
t1 = t17 + t6		C0 0,1			
t2 = t18 – t14		C1 1,0			
t3 = t19 – t15		C1 1,1 / Notify Core 1			
t6 + t6 = t6	Dummy	Wait For Core 1	t8’ = t7’ + t2		
			t9’ = t7’ + t3		
				Fr.(f1,0 + i.(-f1,1))	
			t12’ = t7’ – t9’		
			t13’ = t30’.t8’		
			t14’ = t31’.t12’		
			t15’ = β.t14’		
			t18’=t13’ + t15’		S1,0
			t15’= 13’ + t14’		
			t13’= t30’ + t31’		
			t14’ = t8’ + t12’		
			t13’ = t13’.t14’		
			t19’ =t13’– t15’		S1,1 / Notify Core 0
t20 = t6 + t0		S0,0		DONE	
t21 = t6 – t1		S0,1			
t22 = t6 + t18’		S1,0			
t23 = t6 + t19’		S1,1			
t24 = t30 + t6		Z0,0			
t25 = t6 + t6		Z0,1			
t26 = t6 + t6		Z1,0			
t27 = t6 + t6		Z1,1			
	DONE				

Exponentiation case for bit=’1’

	ACORE 0			ACORE 1	
Micro Code	**Operation**	**Explanation**	**Micro Code**	**Operation**	**Explanation**
t0 = f0,0		BF result	t0’ = Xa		
t1 = f0,1		BF result	t1’ = Ya		
t2 = f1,0		BF result	t2’ = Za		
t3 = f1,1		BF result	t3’ = xq0,0		Real
t4 = Xb			t4’ = xq0,1		Imaginary
t5 = Yb			t5’ = yq1,0		Real
t6 = 0			t6’ = yq1,1		Imaginary
t30 = 1			t7’ = 0		

t31 = β			t28' = a		
			t29' = β		
t20 = S0,0			t30' = Fr	Frobenious	Real
t21 = S0,1			t31' = i.Fr	Frobenious	Imaginary
t22 = S1,0					
t23 = S1,1					
t24 = Z0,0					
t25 = Z0,1					
t26 = Z1,0					
t27 = Z1,1					
	Z = Z^2			**Z = Z^2**	
t6 = t6 + t6	Dummy	Wait For Core 1	t7' = t7' + t7'	Dummy	
			t17' = t7' + t26		
			t18' = t7' + t27		Notify Core 0
t7 = t24.t24	T1 = Z0.Z0		t8' = t17'.t17'	T2 = Z1.Z1	
t8 = t25.t25			t9' = t18'.t18'		
t9 = β.t8			t10' = β.t9'		
t10 = t7 + t9		T1 Real	t19' = t8' + t10'		T2 Real / T3 Imaginary
t9 = t7 + t8			t10' = t8' + t9'		
t7 = t24 + t25			t8' = t17' + t18'		
t8 = t24 + t25			t9' = t17' + t18'		
t7 = t7.t8			t8 ' = t8'.t9'		
t11 = t7 – t9		T1 Imaginary/ Wait For Core 1	t20' = t8' – t10'		T2 Imaginary
			t21' = β.t20'	T3 = γ.T2 = i(t2,0 + i.t2,1)=β.t2,1 + i.t2,0	T3 Real / Notify Core 0
t16= t10+t21'	C0 = T1 + T3	C0 Real	t7' = t7' + t7'	Dummy	Wait For Core 0
t17= t11+t19'		C0 Imaginary			
t14= t10+t19'	T3 = T1 + T2	T3 Real			
t15= t11+t20'		T3 Imaginary			
t10 = t24 + t26	T1 = Z0 + Z1	T1 Real			
t11 = t25 + t27		T1 Imaginary			
t12= t24 + t26	T2 = Z0 + Z1	T2 Real			
t13= t25 + t27		T2 Imaginary			
t7 = t10.t12	T4 = T1.T2				
t8 = t11.t13					
t9 = β.t8					
t18= t7 + t9		T4 Real			
t9 = t7 + t8					
t7 = t10 + t11					
t8 = t12 + t13					
t7 = t7.t8					
t19 = t7 – t9		T4 Imaginary			
t24 = t16 + t6	C1 = T4-T3	Z0 0,0			
t25 = t17 + t6		Z0 0,1			
t26 = t18 – t14		Z1 1,0			
t27 = t19 – t15		Z1 1,1 / Notify Core 1			
t6 = t6 + t6	Dummy	Wait For Core 1	t17' = t7' + t26		
			t18' = t7' + t27		

			t22’ = t7’ + t22		
			t23’ = t7’ + t23		Notify Core 0
	Z = Z.S			**Z = Z.S**	
t7 = t24.t20	T1 = Z0.S0		t8’ = t17’.t22’	T2 = Z1.S1	
t8 = t25.t21			t9’ = t18’.t23’		
t9 = β.t8			t10’ = β.t9’		
t10 = t7 + t9		T1 Real	t19’ = t8’ + t10’		T2 Real / T3 Imaginary
t9 = t7 + t8			t10’ = t8’ + t9’		
t7 = t24 + t25			t8’ = t17’ + t18’		
t8 = t20 + t21			t9’ = t22’ + t23’		
t7 = t7.t8			t8 ‘ = t8’.t9’		
t11 = t7 – t9		T1 Imaginary/ Wait For Core 1	t20’ = t8’ – t10’		T2 Imaginary
			t21’ = β.t20’	T3 = γ.T2 = i(t2,0 + i.t2,1)=β.t2,1 + i.t2,0	T3 Real / Notify Core 0
t16= t10+t21’	C0 = T1 + T3	C0 Real		DONE	
t17= t11+t19’		C0 Imaginary			
t14= t10+t19’	T3 = T1 + T2	T3 Real			
t15= t11+t20’		T3 Imaginary			
t10 = t24 + t26	T1 = Z0 + Z1	T1 Real			
t11 = t25 + t27		T1 Imaginary			
t12= t20 + t22	T2 = S0 + S1	T2 Real			
t13= t21 + t23		T2 Imaginary			
t7 = t10.t12	T4 = T1.T2				
t8 = t11.t13					
t9 = β.t8					
t18= t7 + t9		T4 Real			
t9 = t7 + t8					
t7 = t10 + t11					
t8 = t12 + t13					
t7 = t7.t8					
t19 = t7 – t9		T4 Imaginary			
t24 = t16 + t6	C1 = T4-T3	Z0 0,0			
t25 = t17 + t6		Z0 0,1			
t26 = t18 – t14		Z1 1,0			
t27 = t19 – t15		Z1 1,1			
	DONE				

Exponentiation case for bit=’0’

	ACORE 0			ACORE 1	
Micro Code	**Operation**	**Explanation**	**Micro Code**	**Operation**	**Explanation**
t0 = f0,0			t0’ = Xa		
t1 = f0,1			t1’ = Ya		
t2 = f1,0			t2’ = Za		
t3 = f1,1			t3’ = xq0,0		Real
t4 = Xb			t4’ = xq0,1		Imaginary
t5 = Yb			t5’ = yq1,0		Real
t6 = 0			t6’ = yq1,1		Imaginary
t30 = 1			t7’ = 0		
t31 = β			t28’ = a		
			t29’ = β		
t20 = S0,0			t30’ = Fr	Frobenious	Real

t21 = S0,1			t31' = i.Fr	Frobenious	Imaginary
t22 = S1,0					
t23 = S1,1					
t24 = Z0,0					
t25 = Z0,1					
t26 = Z1,0					
t27 = Z1,1					
	Z = Z^2			**Z = Z^2**	
t6 = t6 + t6	Dummy	Wait For Core 1	t7' = t7' + t7'	Dummy	
			t17' = t7' + t26		
			t18' = t7' + t27		Notify Core 0
t7 = t24.t24	T1 = Z0.Z0		t8' = t17'.t17'	T2 = Z1.Z1	
t8 = t25.t25			t9' = t18'.t18'		
t9 = β.t8			t10' = β.t9'		
t10 = t7 + t9		T1 Real	t19' = t8' + t10'		T2 Real / T3 Imaginary
t9 = t7 + t8			t10' = t8' + t9'		
t7 = t24 + t25			t8' = t17' + t18'		
t8 = t24 + t25			t9' = t17' + t18'		
t7 = t7.t8			t8 ' = t8'.t9'		
t11 = t7 – t9		T1 Imaginary/ Wait For Core 1	t20' = t8' – t10'		T2 Imaginary
			t21' = β.t20'	T3 = γ.T2 = i(t2,0 + i.t2,1)=β.t2,1 + i.t2,0	T3 Real / Notify Core 0
t16= t10+t21'	C0 = T1 + T3	C0 Real		DONE	
t17= t11+t19'		C0 Imaginary			
t14= t10+t19'	T3 = T1 + T2	T3 Real			
t15= t11+t20'		T3 Imaginary			
t10 = t24 + t26	T1 = Z0 + Z1	T1 Real			
t11 = t25 + t27		T1 Imaginary			
t12= t24 + t26	T2 = Z0 + Z1	T2 Real			
t13= t25 + t27		T2 Imaginary			
t7 = t10.t12	T4 = T1.T2				
t8 = t11.t13					
t9 = β.t8					
t18= t7 + t9		T4 Real			
t9 = t7 + t8					
t7 = t10 + t11					
t8 = t12 + t13					
t7 = t7.t8					
t19 = t7 – t9		T4 Imaginary			
t24 = t16 + t6	C1 = T4-T3	Z0 0,0			
t25 = t17 + t6		Z0 0,1			
t26 = t18 – t14		Z1 1,0			
t27 = t19 – t15		Z1 1,1			
	DONE				

Changing the places of variables, for powering t.

	ACORE 0			ACORE 1	
Micro Code	**Operation**	**Explanation**	**Micro Code**	**Operation**	**Explanation**
t0 = f0,0		BF result	t0' = Xa		
t1 = f0,1		BF result	t1' = Ya		

t2 = f1,0		BF result	t2’ = Za		
t3 = f1,1		BF result	t3’ = xq0,0		Real
t4 = Xb			t4’ = xq0,1		Imaginary
t5 = Yb			t5’ = yq1,0		Real
t6 = 0			t6’ = yq1,1		Imaginary
t30 = 1			t7’ = 0		
t31 = β			t28’ = a		
			t29’ = β		
t20 = S0,0			t30’ = Fr	Frobenious	Real
t21 = S0,1			t31’ = i.Fr	Frobenious	Imaginary
t22 = S1,0					
t23 = S1,1					
t24 = Z0,0		s^k1			
t25 = Z0,1		s^k1			
t26 = Z1,0		s^k1			
t27 = Z1,1		s^k1			
t20 = t6 + t0			t7’ = t7’ + t7’	Dummy	Wait For Core 0
t21 = t6 + t1					
t22 = t6 + t2					
t23 = t6 + t3					
t0 = t6 + t24					
t1 = t6 + t25					
t2 = t6 + t26					
t3 = t6 + t27					
t24 = t30 + t6		Z0,0			
t25 = t6 + t6		Z0,1			
t26 = t6 + t6		Z1,0			
t27 = t6 + t6		Z1,1 / Notify Core 1			
	DONE			DONE	

Last operation of exponentiation: $f^{(q^4-1)/r} = s^{k_1} * t^{k_2}$

	ACORE 0			ACORE 1	
Micro Code	**Operation**	**Explanation**	**Micro Code**	**Operation**	**Explanation**
t0 = f0,0		s^k1	t0’ = Xa		
t1 = f0,1		s^k1	t1’ = Ya		
t2 = f1,0		s^k1	t2’ = Za		
t3 = f1,1		s^k1	t3’ = xq0,0		Real
t4 = Xb			t4’ = xq0,1		Imaginary
t5 = Yb			t5’ = yq1,0		Real
t6 = 0			t6’ = yq1,1		Imaginary
t30 = 1			t7’ = 0		
t31 = β			t28’ = a		
			t29’ = β		
t20 = S0,0			t30’ = Fr	Frobenious	Real
t21 = S0,1			t31’ = i.Fr	Frobenious	Imaginary
t22 = S1,0					
t23 = S1,1					
t24 = Z0,0		t^k0			
t25 = Z0,1		t^k0			
t26 = Z1,0		t^k0			
t27 = Z1,1		t^k0			
t6 = t6 + t6	Dummy		t7’ = t7’ + t7’	Dummy	Wait For Core 0
t20 = t6 + t0					

t21 = t6 + t1					
t22 = t6 + t2					
t23 = t6 + t3		Notify Core 1			
t6 = t6 + t6	Dummy	Wait For Core 1	t17' = t7'+ t26		
			t18' = t7'+ t27		
			t22' = t7'+ t22		
			t23' = t7'+ t23		Notify Core 0
	S.T			**S.T**	
t7 = t24.t20	T1 = S0.T0		t8' = t17'.t22'	T2 = S1.T1	
t8 = t25.t21			t9' = t18'.t23'		
t9 = β.t8			t10' = β.t9'		
t10 = t7 + t9		T1 Real	t19'=t8'+ t10'		T2 Real / T3 Imaginary
t9 = t7 + t8			t10' = t8' + t9'		
t7 = t24 + t25			t8'= t17'+ t18'		
t8 = t20 + t21			t9'=t22' + t23'		
t7 = t7.t8			t8 '= t8'.t9'		
t11 = t7 – t9		T1 Imaginary/ Wait For Core 1	t20'=t8' – t10'		T2 Imaginary
			t21' = β.t20'	T3 = γ.T2 = i(t2,0+i.t2,1)=β.t2,1 + i.t2,0	T3 Real / Notify Core 0
t16= t10+t21'	C0 = T1 + T3	C0 Real		DONE	
t17= t11+t19'		C0 Imaginary			
t14= t10+t19'	T3 = T1 + T2	T3 Real			
t15= t11+t20'		T3 Imaginary			
t10 = t24 + t26	T1 = T0 + T1	T1 Real			
t11 = t25 + t27		T1 Imaginary			
t12= t20 + t22	T2 = S0 + S1	T2 Real			
t13= t21 + t23		T2 Imaginary			
t7 = t10.t12	T4 = T1.T2				
t8 = t11.t13					
t9 = β.t8					
t18= t7 + t9		T4 Real			
t9 = t7 + t8					
t7 = t10 + t11					
t8 = t12 + t13					
t7 = t7.t8					
t19 = t7 – t9		T4 Imaginary			
t0 = t16 + t6	C1 = T4-T3	f0 0,0			
t1 = t17 + t6		f0 0,1			
t2 = t18 – t14		f1 1,0			
t3 = t19 – t15		f1 1,1			
	DONE				