Core Based Architecture to Speed Up Optimal Ate Pairing on FPGA Platform

Santosh Ghosh[1,⋆], Ingrid Verbauwhede[1], and Dipanwita Roychowdhury[2]

[1] KU Leuven and IBBT
Dept. Electrical Engineering-ESAT/SCD/COSIC
Kasteelpark Arenberg 10, 3001 Heverlee-Leuven, Belgium
firstname.lastname@esat.kuleuven.be

[2] Indian Institute of Technology Kharagpur
Dept. Computer Science and Engineering
Kharagpur, WB, 721302, India
drc@cse.iitkgp.ernet.in

Abstract. This paper presents an efficient implementation of optimal-ate pairing over BN curves. It exploits the highly optimized IP cores available in modern FPGAs to speed up pairing computation. The pipelined datapaths for $\mathbb{F}_p$-operations and suitable memory cores help to reduce the overall clock cycle count more than 50%. The final design, on a Virtex-6 FPGA, computes an optimal-ate pairing having 126-bit security in 0.375 *ms* which is a 32% speedup from state of the art result.

Keywords: Pairing, BN curves, prime fields, FPGA, Karatsuba, Montgomery, Pipeline, IP core.

1 Introduction

The use of pairings in constructive cryptographic applications are running in their second decade. During this period it has gained a lot of importance because it enables practical realization of numerous protocols. At the same time it is also important to implement pairings for using those protocols in practice. Different alternatives have been derived from the original proposal of Tate pairing for its efficient computation. Optimal-ate pairing [24] is to date the most efficient one computed over elliptic curves (E) defined over a large prime field ($\mathbb{F}_p$). On the other hand, several algebraic curves have been discovered for providing better pairing computation technique as well as for achieving better security. We call them pairing-friendly curves. Barreto-Naehrig curve [3] is the most popular pairing-friendly curve in current days. It is well studied that the optimal-ate pairing on BN curves is one of the best choices of selecting pairings in practice [2].

This paper aims to design an efficient hardware architecture for computing optimal-ate pairing on BN curves. The architecture exploits highly optimized

⋆ The work started when the first author was in IIT Kharagpur, India. Now he is a Postdoctoral Fellow at KU Leuven funded by the IAP Programme P6/26 BCRYPT of Belgian Science Policy (Belspo).

IP cores available for modern FPGAs. The in-built independencies of underlying operations of the pairing computation are fully utilized in order to run an optimized pipeline datapath with reduced number of stall cycles. The memory architecture based on IP cores are efficiently used for generating pipeline operands and storing intermediate results which reduces the use of registers in the design too. The pipelined datapath together with said memory architecture helps to reduce clock cycle count of the pairing computation. A dedicated inversion unit is also incorporated into the design for reducing further cycle count. In total, the final design achieves 32% speedup from the existing premier design [6] for computing optimal-ate pairing.

We start with a brief overview of optimal-ate pairing and its computation procedure over BN curves in § 2. The IP cores that are used in this design are introduced in § 3. The design of the most important underlying $\mathbb{F}_p$-arithmetic block is described in § 4 followed by the description of overall core-based architecture in § 5. The scheduling of operations in order to compute different steps of the pairing algorithm is given in § 6. In § 7, we provide the performance study of the new design with respect to existing results. The paper is concluded in § 8.

2 Optimal-Ate Pairing

Optimal-ate pairing [24] is a non-degenerative bilinear map from $\mathbb{G}_2 \times \mathbb{G}_1$ to $\mathbb{G}_T$ where $\mathbb{G}_2$ and $\mathbb{G}_1$ to be specific subgroups of $E(\mathbb{F}_{p^k})$, and $\mathbb{G}_T$ to be a subgroup of $\mathbb{F}^*_{p^k}$. Let n is a large odd prime dividing $\#E(\mathbb{F}_p)$, and k corresponds to the embedding degree that is the smallest positive integer such that $n|(p^k-1)$. This paper focuses on the optimal-ate pairing on Barreto-Naehrig curve [3], which is well-suited for 128-bit security level and has degree six twist.

A BN curve is an elliptic curve defined over $\mathbb{F}_p$ by following equation.

$$E : y^2 = x^3 + b,$$

where $b \neq 0$ such that $\#E = n$, and $k = 12$. The BN parameters are defined by a suitable $z \in \mathbb{Z}$ such that $p = 36z^4 + 36z^3 + 24z^2 + 6z + 1$ and $n = 36z^4 + 36z^3 + 18z^2 + 6z + 1$ are prime. This paper focuss on optimal-ate pairing with $r = 6z + 2$ defined as [2]:

$$\begin{aligned} a_{opt} : E(\mathbb{F}_{p^{12}}) \cap Ker(\pi_p - p) \times E(\mathbb{F}_p[n]) &\rightarrow \mathbb{F}^*_{p^{12}}/(\mathbb{F}^*_{p^{12}})^n \\ (Q, P) &\mapsto \left(f_{(r,Q)}(P) \cdot g_{(rQ,\pi_p(Q))}(P) \cdot g_{(rQ+\pi_p(Q),-\pi_p^2(Q))}(P)\right)^{(p^{12}-1)/n} \end{aligned}$$

where π_p is the Frobenius map on the curve ($\pi_p(x,y) = (x^p, y^p)$), and $g_{(Q_1,Q_2)}$ is the line through Q_1 and Q_2.

2.1 Computation Procedure

Algorithm 1 computes above optimal-ate pairing. We choose BN curve $E : y^2 = x^3 + 2;\ z = -(2^{62} + 2^{55} + 1) < 0$. The algorithm consists of two major parts

: namely, *Miller's loop* executed in line 2 to line 7, and *final exponentiation* executed in line 12 to line 13. In order to accommodate the negative r, line 8 computes a negation in $\mathbb{G}_2$ to make the final accumulator T the result of $[-|r|]Q$, and the value of $f_{(r,Q)}(P)$ is raised to the power p^6 which is equivalent to f^{-1} as shown in [2]. In line 10 to line 12, the algorithm computes $g_{(rQ,\pi_p(Q))}(P)$ and $g_{(rQ+\pi_p(Q),-\pi_p^2(Q))}(P)$, which are multiplied with f too. With above parameters the addition steps (line 5) invokes only four times throughout the Miller's loop which at the end helps to achieve higher speed of the pairing computation.

Algorithm 1. Optimal-ate pairing on BN curve ($t < 0$)

```
Input: $P = (x_P, y_P) \in E(\mathbb{F}_p[n])$, $Q = (x_Q\gamma^2, y_Q\gamma^3) \in E(\mathbb{F}_{p^{12}}) \cap Ker(\pi_p - p)$
        with $x_Q$ and $y_Q \in \mathbb{F}_{p^2}$, $r = |6t+2| = \Sigma_{i=0}^{s-1} r_i 2^i$.
Output: $a_{opt}(Q,P) \in \mathbb{F}_{p^{12}}$.
 1. $T = (X_T\gamma^2, Y_T\gamma^3, Z_T) \leftarrow (x_Q\gamma^2, y_Q\gamma^3, 1)$, $f \leftarrow 1$ ;
 2. for $i = s-2$ downto 0 do
 3.   $g \leftarrow l_{(T,T)}(P)$, $T \leftarrow 2T$, $f \leftarrow f^2$, $f \leftarrow f \cdot g$ ;
 4.   if $r_i = 1$ then
 5.      $g \leftarrow l_{(T,Q)}(P)$, $T \leftarrow T + Q$, $f \leftarrow f \cdot g$ ;
 6.   endif
 7. endfor
 8. $T \leftarrow -T$, $f \leftarrow f^{p^6}$ ;
 9. $Q_1 \leftarrow \pi_p(Q)$, $Q_2 \leftarrow -\pi_p^2(Q)$ ;
10. $g \leftarrow l_{(T,Q_1)}(P)$, $T \leftarrow T + Q_1$, $f \leftarrow f \cdot g$ ;
11. $g \leftarrow l_{(T,Q_2)}(P)$, $T \leftarrow T + Q_2$, $f \leftarrow f \cdot g$ ;
12. $f \leftarrow \left(f^{p^6-1}\right)^{p^2+1}$ ;
13. $f \leftarrow f^{(p^4-p^2+1)/n}$ ;
14. return $f$ ;
```

Algorithm 1 employs arithmetic in $\mathbb{F}_{p^{12}}$. High-performance arithmetic over extension fields is achieved through a tower of extensions using irreducible binomials [18]. Accordingly, in our targeted setting we represent $\mathbb{F}_{p^{12}}$ using the towering scheme used in [2,22]:

$$\begin{aligned} \mathbb{F}_{p^2} &= \mathbb{F}_p[i]/(i^2 - \beta), \text{ where } \beta = -1. \\ \mathbb{F}_{p^4} &= \mathbb{F}_{p^2}[s]/(s^2 - \xi), \text{ where } \xi = 1 + i. \\ \mathbb{F}_{p^{12}} &= \mathbb{F}_{p^4}[t]/(t^3 - s) \ = \ \mathbb{F}_{p^2}[\tau]/(\tau^6 - \xi). \end{aligned}$$

Throughout the pairing computation we follow the towering $\mathbb{F}_{p^2} \to \mathbb{F}_{p^4} \to \mathbb{F}_{p^{12}}$ as it is shown in [7] that the arithmetic (mainly squaring) in this extension during final exponentiation is much cheaper than other towering extensions. The choice $p \equiv 3 \pmod 4$ accelerates arithmetic in $\mathbb{F}_{p^2}$, since multiplications by $\beta = -1$ can be computed as simple subtractions [22].

3 IP Cores on Xilinx FPGA

Various soft cores, specifically for memory and arithmetic functions, are provided by Xilinx which are easily configured into modern FPGAs like Virtex-6, Virtex-5, or Virtex-4 devices. The Xilinx LogiCORE IP block memory generator [25] core is an advanced memory constructor that generates area and performance-optimized memories using embedded block-RAM (BRAM) resources in Xilinx FPGAs. Available through the core generator software embedded with ISE tool, users can quickly create optimized memories to amend the performance of a design.

Two types of memory cores are used in the current design. Montgomery multiplication (Algorithm 2) uses $P^{(i)} = a^{(i)} \times b^{(i)}$ further for computing the final result ($c^{(i)}$). The value of $P^{(i)}$ is 512-bit long in the current cryptoprocessor. Therefore, we generate a memory core having 512-bit data width. This is a single port memory as its demand of read and write access are exclusive. The current multiplier performs at most 10 Montgomery multiplications in parallel. Thus we generate a memory core having nearest smallest size of 2^4 locations each of which are 512 bits long which is shown in Fig. 1. On the other hand, the top level design integrates two memory cores having 256-bit data width for accommodating one $\mathbb{F}_p$-element in a single memory location. In the current design, the datapath consists of several pipeline stages. Thus in order to avoid pipeline stalls, we generate a 256-bit wide true-dual-port memory core, where both ports are configured independently on the same shared memory space. The usage of this memory core in current design is described in § 5.3.

Similarly, Xilinx LogiCORE IP multiplier [25] implements high-performance, optimized multipliers. It allows the choice of LUTs or dedicated multiplier primitives to be selected for the core implementation. It further provides options for area or speed optimized design. The current design opts for the speed optimization on XtremeDSP slice that consists of dedicated multipliers. Thanks to LogiCORE for permitting a maximum of 64-bit unsigned operands which makes our design more simpler. The maximum speed of the 64-bit IP core is achieved through its 18 pipeline stages. However, the utilization of such pipeline depth is inconvenient for one pairing computation and need to allow several pipeline stalls. Its full utilization is only feasible through hyperthreading technology which in our design can be achieved by sharing the pipeline stages among several pairing computations. At the same time this advanced parallelism makes data-flow more complex and demands adequately large on-chip memory too. The current design tries to make a trade-off among speed, area, and design complexity. It finds that five stage pipeline of a 64-bit multiplier core provides the most suitable design with respect to computing one optimal-ate pairing at a time. With such design choices the IP core achieves a maximum operating clock frequency of 166 MHz on a Virtex-6 FPGA. Throughout the whole design process we preserve this operating frequency and always maintain the register to register combinatorial critical path having lesser delay than the period of above clock. The construction of such a critical-path constrained datapath makes rest of the design more challenging.

4 Base Field Multiplier

Multiplication in base field is the most important operation for computing a cryptographic pairing. In our case it is called $\mathbb{F}_p$-multiplication which can be executed by several techniques. This paper uses a straight forward Montgomery multiplication algorithm. The algorithm is executed by exploiting underlying Karatsuba multiplication for integers and by employing an efficient architecture. For executing extension field operations we always invoke our only multiplier for generating reduced result for each $\mathbb{F}_p$-multiplication. To speed up extension field arithmetic sometimes a lazy reduction technique is used [6]. However, instead of lazy reduction, our new multiplier executes multiple simultaneous $\mathbb{F}_p$-multiplications on pipelined datapath which ultimately speed up the overall pairing computation.

4.1 Montgomery and Karatsuba Combination

Montgomery multiplication algorithm avoids the division by p. The finite field multiplication is performed as modulo 2^n having $n = \lceil \log_2 p \rceil$ instead of modulo p. It is necessary to convert each operand from integer to its equivalent Montgomery form which costs another Montgomery multiplication. However, for repeated multiplications used in a pairing computation it is sufficient to convert the operands once at the beginning which is converted back at the end.

The Montgomery multiplication algorithm for large characteristic field is shown in Algorithm 2. The parenthesized indices represent the variables associated with that instruction. The indices are mainly used to identify an instruction and its associate variables inside our pipeline architecture. Algorithm 2 consists of three n bit integer multiplications, which determines the overall efficiency of the algorithm. Here we propose an efficient Montgomery multiplier architecture for modern FPGAs. Highly optimized IP cores available for FPGA devices together with our careful datapath design help to achieve an efficient pipelined architecture for Montgomery multiplication.

Algorithm 2. Montgomery multiplication

```
Input: $M = p$; $n = \lceil \log_2 M \rceil$; $R = 2^n$; $M' = -M^{-1} \bmod R$; $a^{(i)}, b^{(i)} \in \mathbb{Z}_M$.
Output: $a^{(i)} \cdot b^{(i)} \cdot R^{-1} \bmod M$.
1. $P^{(i)} \leftarrow a^{(i)} \cdot b^{(i)}$ ;
2. $U^{(i)} \leftarrow (P^{(i)} \bmod R) \cdot M' \bmod R$ ;
3. $c^{(i)} \leftarrow (P^{(i)} + U^{(i)} \cdot M)/R$ ;
4. if $c^{(i)} \geq M$ then
5.     $c^{(i)} \leftarrow c^{(i)} - M$ ;
6. return $c^{(i)}$;
```

Figure 1 depicts the proposed Montgomery multiplier architecture. It consists of a 256×256 bit Karatsuba multiplier, which is constructed by nine 64×64 bit multiplier cores. There is a small memory unit for holding intermediate result of Montgomery multiplication which are used in later steps. Main novelty of the current design lies to efficient utilization of in-built multiplier and memory cores to achieve an optimized design on a modern FPGA platform. The top level of the architecture computes three integer multiplications in serial. The result of third multiplication is added with the result of the first one followed by a optional reduction (subtraction) to compute the result of a Montgomery multiplication. Although, the proposed design consists of a pipeline structure which is able to compute more than one multiplication in parallel. The detailed construction and its functionality is described in following sections.

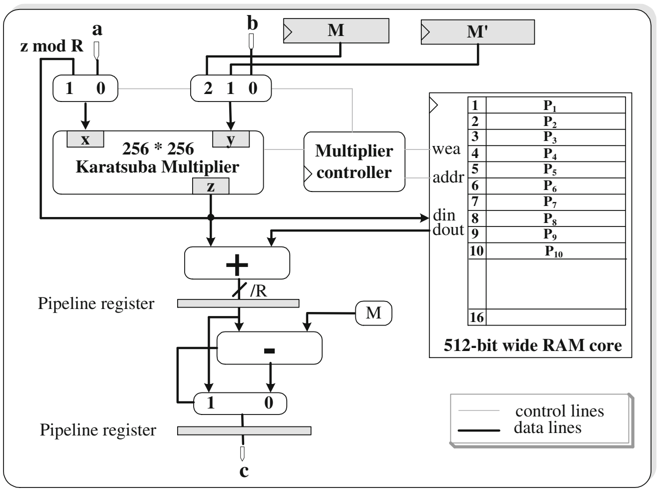

Fig. 1. The Montgomery multiplier

4.2 Delay Constrained Design

The proposed integer multiplier follows Karatsuba technique for performing 256-bit multiplications. Thus, three 128-bit multiplications, each of which is computed by three 64-bit multiplier cores, are performed in parallel. The post-multiplier operations are put into one additional pipeline stage for generating an 128×128 multiplication result. However, we find that the delay of datapath of post-multiplier operations is in between one and two clock periods for getting a 256×256 bit multiplication results. Thus, it is broken into two parts − adds two more pipeline stages. On the other hand, pre-multiplier datapath consists of an input multiplexer, an 128-bit adder and a 64-bit adder circuits, which forms two more pipeline stages. To sum up, the whole multiplier consists of 10 pipeline stages on which 10 independent multiplications can be executed in parallel.

The Montgomery multiplication algorithm (Algorithm 2) consists of three dependent integer multiplications. Therefore, we explore the parallelism at finite field level for which 10 independent $\mathbb{F}_p$-multiplications are fetched and issued in parallel. The proposed design performs each Montgomery multiplication by executing operations divided in following five steps.

1. compute $P^{(i)} = a^{(i)} \times b^{(i)}$
2. store $P^{(i)}$ in RAM, and compute $U^{(i)} = (P^{(i)} \bmod R) \times M^{'}$
3. compute $V^{(i)} = (U^{(i)} \bmod R) \times M$
4. compute $c^{(i)} = (P^{(i)} + V^{(i)})/R$
5. compute $c^{(i)} = c^{(i)} - M$, if $c^{(i)} \geq M$.

We schedule the computation of $P^{(i)}$, $1 \leq i \leq 10$ first into the pipeline then all $U^{(i)}s$ followed by 10 $V^{(i)}s$. As soon as a $P^{(i)}$ gets out from the pipeline it is scheduled on-the-fly for computing $U^{(i)}$ as defined in step 2. The $P^{(i)}s$ are also stored into the 512-bit wide single-port-RAM (shown in Fig. 1) to use it further in step 4. Except $P^{(i)}$ it is not necessary to store other intermediate results $(U^{(i)}, V^{(i)})$. They are scheduled on-the-fly for further processing. The 31-st clock onwards from the beginning we start to receive $V^{(i)}$, which are then processed by two additional steps (step 4 and step 5) in two consecutive clock cycles. Therefore, to sum up, the cost of 10 Montgomery multiplications is 42 clock cycles in the current design. During these 42 clock cycles the multiplier communicates with external memory only at the first 10 clock cycles (to read $a^{(i)}$ and $b^{(i)}$) and the last 10 clock cycles (to write $c^{(i)}$). In between these two 10 clock cycles periods there are remaining 22 clock cycles when the external memory is free to access for other operations. These free cycles are utilized to accumulate $\mathbb{F}_p$ multiplication results to produce results in extension fields, to perform constant multiplications, and to perform other intermediate operations in pairing computation. This two levels of parallelism, namely, multiple $\mathbb{F}_p$-multiplications on a single unit and several $\mathbb{F}_p$-operations on different units, help to speed up pairing computation on the proposed design.

5 Architecture for Pairing

As shown in Algorithm 1, the pairing computation consists of following major operations.

1. *Doubling step:* An elliptic curve point doubling operation together with the computation of line function g.
2. *Addition step:* An elliptic curve point addition and the computation of g.
3. *Squaring:* Squaring of Miller variable f.
4. *Sparse multiplication:* A multiplication of Miller variable f with g having only half of the non-zero coefficients.
5. *Frobenius and Easy exponentiation:* Intermediate operations of Miller loop and hard exponentiation.
6. *Hard exponentiation:* Powering the intermediate result by $\phi_i(x)/n$ in cyclotomic subgroup.

In optimal-ate pairing on BN curve, first two steps are performed in $\mathbb{F}_{p^2}$, and most of the operations in other steps are performed in $\mathbb{F}_{p^{12}}$. Several advanced techniques can compute these extended field operations with much lower costs compared to their straight forward computation [8,16]. We choose the techniques having lower number of multiplications and squarings. The underlying operations in each techniques are computed in the base field. Therefore, we visualize the whole pairing computation as a sequence of $\mathbb{F}_p$ operations and try to execute them as fast as possible on a target platform.

5.1 Overview of the Architecture

The cost of a pairing computation is normally represented by the number of base field multiplications [17]. However, it is observed that apart from multiplications, a pairing computation consists of huge number of additions, subtractions and constant-multiplications. In current days the costs of a multiplication and an addition/subtraction with respect to time is almost same. Thus, the cost of a pairing computation equivalently depends on the efficiency of the "architecture" of all such operations. Moreover, this cost varies with the efficiency of the "scheduling" technique used on a specific implementation. Therefore, throughout the implementation we give equal attention to both architecture design and scheduling which in together maximizes the utilization of individual components and finally speeds up the pairing computation with constrained resources.

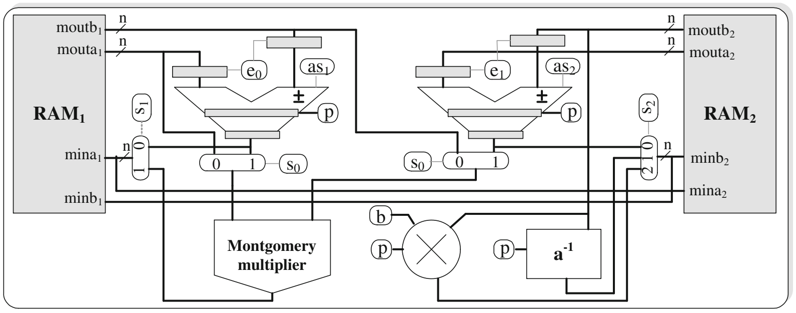

Fig. 2. The architecture for pairing computation

Figure 2 depicts the datapath of the architecture for pairing computation. It consists of a multiplier, two adder/subtractors, a constant-multiplier, and an inversion unit. All of them can independently perform respective operations in $\mathbb{F}_p$. In order to maximize their utilization we incorporate two true-dual-port RAM cores (call them RAM_1 and RAM_2) each of which contains identical data during a pairing computation. The operations in extension fields need to execute independent multiplications like $a \times b$ and $(a \pm b)(c \pm d)$. In order to support them without any pipeline stall, each of the multiplier inputs is multiplexed between an output port of RAM_1 and an output of adder/subtractor ($\pm$ block). The architecture facilitates the port configuration in such a way that the output of each of functional units can be written in the same address of both RAMs in parallel. This helps to keep the identical data in both memory cores throughout the pairing computation which are exploited to improve the degree of parallelism.

5.2 Architecture Details

The architecture is developed with several pipeline stages in each of the functional units. Number of pipeline stages are identified to meet the maximum operating frequency provided by the 64×64 multiplier core as described in § 3.

Modular Adder Subtractor. The addition and subtraction in $\mathbb{F}_p$ can be realized by two consecutive n-bit adder circuits which produce final result in only one clock cycle. However, the latency of such a circuit in Virtex-6 FPGA is $11ns$, which is 1.8 times of our target critical path. We therefore divide this datapath into two pipeline stages which is illustrated in Fig. 3. The whole design now demands 130 Virtex-6 FPGA slices on which it achieves a maximum operating frequency of 183 MHz. Due to the pipeline structure its throughput is one $\mathbb{F}_p$ addition/subtraction per clock.

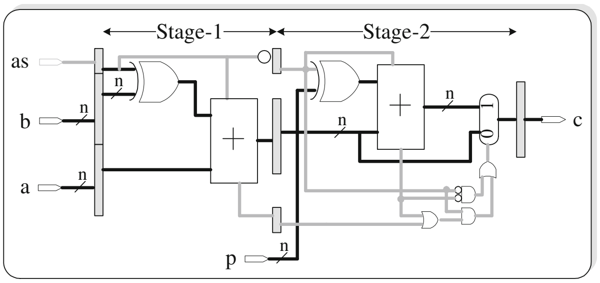

Fig. 3. Two stage pipeline for $\mathbb{F}_p$ addition and subtraction

Constant Multiplier. There are some operations in doubling and addition steps where a finite field element ($a \in \mathbb{F}_p$) is multiplied with small integers (≤ 6). We develop an adder based five stage pipeline structure for constant-multiplications which executes the target operations by following an addition chain. The first pipeline stage performs $2a$ mod p, where doubling is simple rewiring followed by a conditional subtraction. Second and third stages is formed by following modular adder/subtractor (Fig. 3) unit. The only difference is that it performs both $3a = (2a + a)$ mod p and $4a = 2 \times 2a$ mod p in parallel. The second stage performs addition and doubling whereas we use the third stage for their reductions. The results of $3a$ and $4a$ are produced at the end of third stage. Similarly, fourth and fifth stages are formed to execute $5a = (2a+3a)$ mod p and $6a = 2 \times 3a$ mod p. The pipeline registers are designed with optimum storage space. For example, after second stage the value of a is no longer being used, so pipeline does not carry it beyond this point. Similarly, $4a$ is never used in further pipeline stages and the values of $2a$ and $3a$ are last used in the fourth stage. Through such observations, the pipelined constant-multiplier is designed, which optimizes overall area as well as time. Respective life-time diagram is shown in Fig. 4.

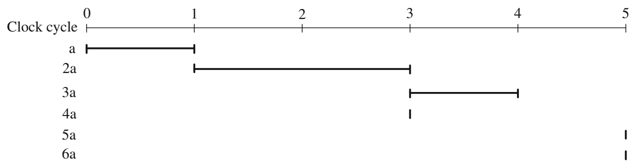

Fig. 4. Life time diagram of constant multiplication

Inversion Block. This block is developed as a dedicated unit for performing inversion in $\mathbb{F}_p$. It is based on the Extended Euclidean Algorithm. This unit is rarely (only once) used for computing a pairing. The functionality of this is described in § 6.4.

5.3 True Dual Port RAM

At this design stage we have already customized the datapath for pairing computation. So further speedup could be gained through the maximization of datapath utilization. A basic requirement for computing any two-input, one-output operation is to have two operands in parallel at the input ports of respective unit and an output destination available. This motivates the use of true-dual-port RAM for current design. It is called *true-dual-port* because both ports can independently perform read/write operations on the same shared memory space.

This RAM core is generated through Xilinx LogiCORE IP block generator tool (introduced in § 3). In the current design it is configure in write first mode having register at output port of the memory. Due to which we allow one clock cycle delay between address generation and availability of data at respective output port. At the same time, this register separates the datapath through multiplexer inside the memory block from the datapath between memory ports and the beginning of first pipeline stage of a functional unit. Otherwise, this combined datapath becomes longer than our target critical path. Each of this two memory cores contains 2^9 locations having 256-bit width that is sufficient to hold local and global variables during a pairing computation.

5.4 Working Principle of the Architecture

The overall architecture is constructed by observing that the pairing computation has several sets of independent base field operations. We perform an in-depth analysis on optimal-ate pairing algorithm to optimize such instruction sets in order to maximize the utilization of the customized datapath with minimum storage space for temporary results. The analysis suggests that the formation of such instruction sets each of which containing at most 10 base field multiplications can utilize the current multiplier with minimum stall cycles for computing a single optimal-ate pairing. We call them *opt_set*. It is already described in § 5.1 that our architecture generates the result of $(a \pm b)$ and $(c \pm d)$ on-the-fly for performing $(a \pm b)(c \pm d)$. Thus multiplications in this form are also counting as a simple $\mathbb{F}_p$-multiplication during formation of the *opt_set*.

The execution of such an i-th *opt_set* on our architecture is as follows:

- It first schedules 10 multiplications on the pipelined multiplier in 10 consecutive clock cycles.
- From 11-*th* clock cycle, it schedules the additions, subtractions, and constant-multiplications on two adder/subtractor units and the constant-multiplier such that their results are written back to the memory within 31-st clock cycle. We perform those operations in this phase such that all multiplication results of $(i-1)$-*th* set are properly utilized and they are no longer used in future. The operations to prepare operands for multiplications of $(i+1)$-th *opt_set* are computed too in this phase.
- The results of the multiplications are available at multipliers output port from 32-nd clocks. These results are written back to the specific 10 conjugative locations in both RAMs from which the multiplications results of $(i-1)$-th *opt_set* are already utilized.

The execution of such a set takes 42 clock cycles, after which a new set is normally scheduled immediately from the next clock. Remember that all memory write operations in this implementation are performed in both RAMs (shown in Fig. 2 by $mina_1/mina_2$ and $minb_1/minb_2$) in same address for achieving higher degree of parallelism.

6 Scheduling and Pairing Computation

The execution control and the scheduling of operations on different functional units are performed by a state machine and few small counter logics. Here we present the instruction set formations for executing every step of the pairing computation. In the current design, the addition costs are hidden to multiplier cycles and therefore we use the techniques for internal operations especially, for extension-field arithmetic, having lower multiplications and squarings.

6.1 Execution of Doubling Step and f^2

There is no dependency between *doubling step* and f^2 computation which are therefore scheduled together. The step-by-step computation of the doubling step and f^2 is provided in Algorithm 5 of A.2. The formula of doubling step is followed from the state of the art existing pairing implementations [2,6,7]. We made rearrangements of the computations for making it suitable for our design. On the other hand, it is shown in [16] that the representation of f in tower extension $\mathbb{F}_{((p^2)^2)^3}$ helps to reduce the operation count of f^2 computation in final exponentiation. Though this towering does not help to reduce the computation costs of f^2 within the Miller loop, but for simplicity, throughout the implementation we use the same towering to represent f.

The operations in this steps as well as other parts of the pairing computation described in this paper are performed either in $\mathbb{F}_{q^2}$ or in $\mathbb{F}_{q^3}$. Various technique for computing multiplication and squaring in such quadratic and cubic extension fields are explained in [8]. In this paper, we follow Karatsuba technique for computing both multiplication and squaring in $\mathbb{F}_{q^3}$, whereas, in case of $\mathbb{F}_{q^2}$ we use Karatsuba technique for multiplication and complex method for squaring. Formula for all such used techniques are provided in A.1. We represent the Miller variable f as :

$$\begin{aligned} f &= f_0 + f_1\tau + f_2\tau^2 + f_3\tau^3 + f_4\tau^4 + f_5\tau^5 \\ &= (f_0 + f_3 s) + (f_1 + f_4 s)t + (f_2 + f_5 s)t^2, \end{aligned}$$

which is considered as: $a_0 + a_1 t + a_2 t^2$ with $a_j \in \mathbb{F}_q, q = p^4$, $0 \leq j \leq 2$. Computation of f^2 in this towering extension consists of 36 multiplications in $\mathbb{F}_p$, which all are independent – though some of them need a few prior additions. On the other hand the computation of doubling step in Projective coordinate requires 27 multiplications in $\mathbb{F}_p$, which are not free to schedule at any point of time as they have several data dependencies. Thanks to our pipeline and memory architecture that we can manage all operations of this phase in 7 *opt_sets*. Among them first *opt_sets* containing 10 multiplications, second one consists of 8 multiplications and each of the remaining five consist of 9 multiplications. After receiving the multiplication results of final *opt_set* a few additions are performed for final accumulation.

6.2 The Addition Step

The addition step consists of 41 $\mathbb{F}_p$-multiplications which have several data dependencies. We compute them by forming five *opt_sets* with few intermediate additions during which the multiplier pipeline runs with bubbles. That is, we do not start $(i + 1)$-th *opt_set* immediately after completing the execution of i-th *opt_set*. However, due to the dual adder/subtractor units these stall cycles are small compared to overall execution cycles. The formula for computing this step in Projective coordinates is provided in [2,6,7]. Algorithm 6 in A.2 provides the same with little rearrangements of operations to fit our current scheduling.

6.3 Computation of $f \cdot g$

The Karatsuba technique costs 54 $\mathbb{F}_p$-multiplications for computing a multiplication in $\mathbb{F}_{((p^2)^2)^3}$. However, in the $f \cdot g$ computations of steps 3, 5, 10, and 11 of Algorithm 1 only half of the coefficients of g are non-zero. Due to the sparse value[1] of g, $f \cdot g$ consists of 39 $\mathbb{F}_p$-multiplications. The respective technique is provided in Algorithm 3 of A.2. We accommodate them in four *opt_sets*, where except the last one each *opt_set* contains 10 base field multiplications. A few additions, which depend on the multiplication results of final *opt_set*, are performed and update the value of f at respective locations at the end.

6.4 Inversion in $\mathbb{F}_p$

For powering f by $p^6 - 1$ in step 12 of Algorithm 1 it is essential to compute an inversion in $\mathbb{F}_{p^{12}}$, which is easily deduced as a single inversion in $\mathbb{F}_p$ along with several multiplications. The inverse of $a \in \mathbb{F}_p$ is in general computed by two methods – *Fermat's Little Theorem* or *Extended Euclidean Algorithm* (EEA). The first one computes inversion through exponentiation $a^{-1} \equiv a^{p-2} \bmod p$. On the other hand an efficient variant of EEA for $\mathbb{F}_p$-inversion is known as *Binary Inversion Algorithm*, which is primarily based on *gcd* computation. The exponentiation is efficiently implemented through an iterated square-and-multiply procedure for which an efficient implementation of the field multiplier is sufficient. However in our pipelined multiplier, execution of a single exponentiation is too costly as its i-th iteration cannot be started before completing $(i-1)$-th iteration. Thus, it will costs $33\lceil \log_2 p \rceil$ clock cycles with right-to-left execution.

On the other hand, an efficient implementation of binary inversion algorithm, as shown in [13], takes $2\lceil \log_2 p \rceil$ clock cycles. The stand alone implementation of this inversion unit requires 1350 Virtex-6 slices. On the other hand without this unit the current design takes $33\lceil \log_2 p \rceil$ number of clock cycles, which is 16.5 times more than the time taken by dedicated inversion unit. Thus we incorporate it into our design especially for computing a single inversion in final exponentiation. With our parameter settings without this unit the current design requires $7,874$ additional clock cycles for computing an inversion.

6.5 Exponentiation by $|z|$

After executing step 12 of Algorithm 1, the value of f becomes an element of the cyclotomic subgroup ($\mathbb{G}_{\phi_{12}(p)}$) in $\mathbb{F}_{p^{12}}$. An efficient technique used in this design for computing step 13 of Algorithm 1 (hard part of final exponentiation) is given in [23]. There are three exponentiations in $\mathbb{G}_{\phi_{12}(p)}$ by $|z|$ which are the most costly operations in this step. With our towering representation this squaring (Algorithm 4 in A.2) is much cheaper than a squaring computed in Miller's loop [16]. This squaring is executed by two *opt_sets* and few final additions and constant multiplications by our design. The whole exponentiation is performed by standard left-to-right square-and-multiply algorithm. Therefore, the multiplication is performed only if the respective exponent bit is one. This multiplication is a full multiplication (having no sparse operands) in $\mathbb{F}_{p^{12}}$, which consists of 54

[1] An operand in $\mathbb{F}_{p^{12}}$ is sparse when some of its coefficients are trivial (i.e., either zero or one).

independent $\mathbb{F}_p$-multiplications. We schedule them on the pipelined multiplier by forming six *opt_sets*.

In contrasts to pipelined design of [6] the current design uses MSB first method. Due to the low Hamming weight of $|z|$ the multiplications cost is vary low compared to the costs of squarings, and the current pipeline is suitable to execute one individual non-linear operation in $\mathbb{F}_{p^{12}}$. On the other hand, [2] shows a compressed technique for exponentiation by $|z|$ using Montgomery's simultaneous inversion trick [20]. However, this technique does not help to speed up pairing computation in our design as an inversion is 127 times slower than a multiplication in the current design.

7 Results

The whole design has been done in Verilog (HDL). Implementation has been performed on Xilinx ISE Design Suit 12.4. Table 1 shows the implementation results. On a Virtex-6 xc6vlx240t-3ff1759 FPGA the proposed design runs at a maximum frequency of $166MHz$. In total, with dedicated inversion unit, this design uses 5163 logic slices, 144 DSP slices and 21 BRAMS. It finishes computation of one 126-bit secure optimal-ate pairing in $375\mu s$. Table 2 gives the clock cycle counts required by the proposed design to computing different steps of an optimal-ate pairing on 126-bit secure BN curve.

Table 1. Area utilization on Virtex-6 FPGA

Current design	**Frequency** $[MHz]$	**Multipliers**	**Logic Elements**	**Memory**
with inversion	166	144 DSP48E1s	5163 slice‡	21 RAMB36E1
without inversion			3813 slice‡	
‡ : One Virtex-6 slice consists of four LUTs and eight flip-flops.				

Table 2. Cycle count for different steps of optimal-ate pairing on BN_{126} curve

Current design	$2T$, $g_{(T,T)}(P)$, and f^2	$T+Q$ and $g_{(T,Q)}(P)$	$f.g$	**Miller's Loop**	a^{-1} in $\mathbb{F}_p$	m^t in $\mathbb{G}_{\phi_k(p)}$	**Post M. Loop**	**Total**
with inv.	314	235	192	$34,092$	508	$7,018$	$28,074$	$62,166$
without inv.	314	235	192	$34,092$	$8,448$	$7,018$	$36,014$	$70,106$

7.1 Comparison with Recent Designs

Table 3 shows the comparative analysis of recent hardware and software results of pairing. With respect to latency of a pairing computation over BN curves with similar security level the present design achieves 32% speedup from the existing premier design proposed in [6]. Its slice counts is also relatively less with cost of more parallelism on higher number of DSP blocks. The clock cycle count of the current design is reduced drastically due to higher parallelism on the pipelined datapath. In contrary the implementation of pairings over general elliptic curves having 128-bit security is still slower than that over a supersingular curves proposed in [14]. This may be due to the easier binary field arithmetic.

Table 3. Performance of hardware and software results of pairings

Designs	**Curve**	**FPGA**	**Area**	**Freq.** $[MHz]$	**Cycle** $[\times 10^3]$	**Delay** $[\mu s]$
This work (inv)	BN_{126}	xc6vlx240t-3	5163 Slices, 144 DSP	166	62	375
(without inv)	BN_{126}	xc6vlx240t-3	3813 Slices, 144 DSP	166	70	422
Cheung *et al.* [6]	BN_{126}	xc6vlx240t-2	7032 Slices, 32 DSP	250	143	573
	BN_{192}	Stratix-III	9910 A, 171 DSP	131	790	6030
Fan et al. [12]	BN_{128}	xc6vlx240t-3	4014 Slices, 42 DSP	210	245	1170
Ghosh *et al.* [15]	BN_{128}	xc4vlx200-12	52000 Slices	50	821	16400
Kammler *et al.* [19]	BN_{128}	130nm CMOS	97000 Gates	338	5,340*	15800
Ghosh *et al.* [14]	$E/\mathbb{F}_{2^{1223}}$	xc6vlx130t-3	15167 Slices	250	76†	190
Estibals [10]	$E/\mathbb{F}_{3^{5\cdot 97}}$	xc4vlx200-11	4755 Slices	192	429	2227
Aranha *et al.* [1]	$Co/\mathbb{F}_{2^{367}}$	xc4vlx25-11	4518 Slices	220	774*	3518
Naehrig *et al.* [21]	BN_{128}	core2 Q6600	–	2394	4,470	1860
Beuchat *et al.* [4]	BN_{126}	core i7 2.8GHz	–	2800	2,330	830
Aranha *et al.* [2]	BN_{126}	Phenom II	–	3000	1,562	520
Aranha *et al.* [1]	genus-2	Core i5	–	2530	2,440	960

† Estimated by the authors. * Estimation provided in [6].

Till 2010, the software for pairing outperforms the hardware and it was a bit uncomfortable to the hardware world. It was due to several unexplored in-built features available in the hardware platforms, especially FPGA platforms for pairing computation. However, at the end of last year it becomes true by the design shown in [6,14] for pairing too that customized hardware always outperforms a pure software. The current design in that respect not only gains the speedup from existing design but also it shows a direction for further improvement of pairing computations through exploitation of several highly optimized IP cores in different platforms.

8 Conclusion

In this paper we have proposed a core based architecture for pairing computation on general elliptic curves defined over large prime fields. Due to intelligent pipeline the proposed design has achieved a 32% speedup over existing designs. Moreover, a dedicated field inversion unit has reduced the clock cycle count of final exponentiation as well as a full pairing computation. The application of IP cores with more pipeline-depth may be targeted in future for executing multiple pairing computations at a time in order to handle several parallel client requests.

Acknowledgements. This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy) and by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

References

1. Aranha, D.F., Beuchat, J.L., Detrey, J., Estibals, N.: Optimal Eta pairing on supersingular genus-2 binary hyperelliptic curves. Cryptology ePrint Archive, Report 2010/559, `http://eprint.iacr.org/`
2. Aranha, D.F., Karabina, K., Longa, P., Gebotys, C.H., López, J.: Faster Explicit Formulas for Computing Pairings over Ordinary Curves. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 48–68. Springer, Heidelberg (2011)
3. Barreto, P.S.L.M., Naehrig, M.: Pairing-Friendly Elliptic Curves of Prime Order. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 319–331. Springer, Heidelberg (2006)
4. Beuchat, J.-L., González-Díaz, J.E., Mitsunari, S., Okamoto, E., Rodríguez-Henríquez, F., Teruya, T.: High-Speed Software Implementation of the Optimal Ate Pairing over Barreto–Naehrig Curves. In: Joye, M., Miyaji, A., Otsuka, A. (eds.) Pairing 2010. LNCS, vol. 6487, pp. 21–39. Springer, Heidelberg (2010)
5. Beuchat, J.L., Detrey, J., Estibals, N., Okamoto, E., Rodríguez-Henríquez, F.: Fast architectures for the η_T pairing over small-characteristic supersingular elliptic curves. IEEE Transactions on Computers 60(2) (2011)
6. Cheung, R.C.C., Duquesne, S., Fan, J., Guillermin, N., Verbauwhede, I., Yao, G.X.: FPGA Implementation of Pairings Using Residue Number System and Lazy Reduction. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 421–441. Springer, Heidelberg (2011)
7. Costello, C., Lange, T., Naehrig, M.: Faster Pairing Computations on Curves with High-Degree Twists. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 224–242. Springer, Heidelberg (2010)
8. Devegili, A., ÓhÉigeartaigh, C., Scott, M., Dahab, R.: Multiplication and squaring on pairing-friendly fields. Cryptology ePrint, Report 2006/471 (2006)
9. Duquesne, S., Guillermin, N.: A FPGA pairing implementation using the residue number system. Cryptology ePrint Archive, Report 2011/176 (2011), `http://eprint.iacr.org/`
10. Estibals, N.: Compact Hardware for Computing the Tate Pairing over 128-Bit-Security Supersingular Curves. In: Joye, M., Miyaji, A., Otsuka, A. (eds.) Pairing 2010. LNCS, vol. 6487, pp. 397–416. Springer, Heidelberg (2010)
11. Fan, J., Vercauteren, F., Verbauwhede, I.: Faster $\mathbb{F}_p$-Arithmetic for Cryptographic Pairings on Barreto-Naehrig Curves. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 240–253. Springer, Heidelberg (2009)
12. Fan, J., Vercauteren, F., Verbauwhede, I.: Efficient Hardware Implementation of $\mathbb{F}_p$-arithmetic for Pairing-Friendly Curves. IEEE Trasaction on Computers (2011), `http://dx.doi.org/10.1109/TC.2011.78`
13. Ghosh, S., Mukhopadhyay, D., Roychowdhury, D.: Petrel: power and timing attack resistant elliptic curve scalar multiplier based on programmable arithmetic unit. IEEE Trans. on Circuits and Systems I 58(11), 1798–1812 (2011)

14. Ghosh, S., Roychowdhury, D., Das, A.: High Speed Cryptoprocessor for η_T Pairing on 128-bit Secure Supersingular Elliptic Curves over Characteristic Two Fields. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 442–458. Springer, Heidelberg (2011)
15. Ghosh, S., Mukhopadhyay, D., Roychowdhury, D.: High Speed Flexible Pairing Cryptoprocessor on FPGA Platform. In: Joye, M., Miyaji, A., Otsuka, A. (eds.) Pairing 2010. LNCS, vol. 6487, pp. 450–466. Springer, Heidelberg (2010)
16. Granger, R., Scott, M.: Faster Squaring in the Cyclotomic Subgroup of Sixth Degree Extensions. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 209–223. Springer, Heidelberg (2010)
17. Hankerson, D., Menezes, A., Scott, M.: Software implementation of pairings. Cryptology and Info. Security Series, ch. 12, pp. 188–206. IOS Press (2009)
18. IEEE: P1363.3: Standard for Identity-Based Cryptographic Techniques using Pairings (2006), `http://grouper.ieee.org/groups/1363/IBC/submissions/`
19. Kammler, D., Zhang, D., Schwabe, P., Scharwaechter, H., Langenberg, M., Auras, D., Ascheid, G., Mathar, R.: Designing an ASIP for Cryptographic Pairings over Barreto-Naehrig Curves. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 254–271. Springer, Heidelberg (2009)
20. Montgomery, P.: Speeding the Pollard and Elliptic Curve Methods of Factorization. Mathematics of Computation 48, 243–264 (1987)
21. Naehrig, M., Niederhagen, R., Schwabe, P.: New Software Speed Records for Cryptographic Pairings. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 109–123. Springer, Heidelberg (2010)
22. Pereira, G.C.C.F., Simplício Jr., M.A., Naehrig, M., Barreto, P.S.L.M.: A Family of Implementation-Friendly BN Elliptic Curves. Journal of Systems and Software 84(8), 1319–1326 (2011)
23. Scott, M., Benger, N., Charlemagne, M., Dominguez Perez, L.J., Kachisa, E.J.: On the Final Exponentiation for Calculating Pairings on Ordinary Elliptic Curves. In: Shacham, H., Waters, B. (eds.) Pairing 2009. LNCS, vol. 5671, pp. 78–88. Springer, Heidelberg (2009)
24. Vercauteren, F.: Optimal pairings. IEEE Transactions on Information Theory 56(1), 455–461 (2010)
25. Xilinx. LogiCORE IP Block Generator (2010), `http://www.xilinx.com`

A Appendix

A.1 Multiplication and Squaring in $\mathbb{F}_{q^2}$ and $\mathbb{F}_{q^3}$

Let an element $\alpha \in \mathbb{F}_{q^2}$ be represented as $\alpha_0 + \alpha_1 X$, where $\alpha_0, \alpha_1 \in \mathbb{F}_q$ and X is an indeterminate. The formula of Karatsuba multiplication $c = ab$ on $\mathbb{F}_{q^2}$ is : $v_0 = a_0 b_0,\ v_1 = a_1 b_1,\ c_0 = v_0 + \zeta v_1,\ c_1 = (a_0 + a_1)(b_0 + b_1) - v_0 - v_1$, where $v_0, v_1, c_0, c_1, a_0, a_1, b_0, b_1 \in \mathbb{F}_q$. Here ζ is a quadratic non-residue in $\mathbb{F}_q$. The cost of such a multiplication is $(3m + 5a + 1\zeta_m)$ in $\mathbb{F}_q$. Similarly, the squaring $c = a^2$ on $\mathbb{F}_{q^2}$ using Complex method is computed by : $v_0 = a_0 a_1,\ c_0 = (a_0 + a_1)(a_0 + \zeta a_1) - v_0 - \zeta v_0,\ c_1 = 2v_0$. The cost of such a squaring is $(2m + 5a + 2\zeta_m)$ in $\mathbb{F}_q$. The equation of c_0 is easily deduced to $a_0^2 + \zeta a_1^2$, which eliminates additions but needs two squaring instead of one multiplication. In the current design squaring and multiplication is performed by same unit with same cost. On the other hand the addition costs are hidden to multiplication costs, thus we use above formula to compute squaring in $\mathbb{F}_{q^2}$.

Similarly, let an element $\alpha \in \mathbb{F}_{q^3}$ be represented as $\alpha_0 + \alpha_1 X + \alpha_1 X^2$, where $\alpha_i \in \mathbb{F}_q$ and X is an indeterminate. The formula of Karatsuba multiplication $c = ab$ on $\mathbb{F}_{q^3}$ is : $v_0 = a_0 b_0,\ v_1 = a_1 b_1,\ v_2 = a_2 b_2,\ c_0 = v_0 + \vartheta((a_1+a_2)(b_1+b_2) - v_1 - v_2),\ c_1 = (a_0+a_1)(b_0+b_1) - v_0 - v_1 + \vartheta v_2,\ c_2 = (a_0+a_2)(b_0+b_2) - v_0 + v_1 - v_2$, where $v_i, c_i, a_i, b_i, \in \mathbb{F}_q$. Here ϑ is a cubic non-residue in $\mathbb{F}_q$. The cost of such a multiplication is $(6m + 15a + 2\vartheta_m)$ in $\mathbb{F}_q$. This multiplication formula is also used for squaring $c = a^2$ on $\mathbb{F}_{q^3}$ replacing b by a. Thus the cost estimation for squaring replaces 6 multiplications by six squaring in $\mathbb{F}_q$.

A.2 Sub-Routines for Optimal-Ate Pairing

Algorithm 3. Computation of $f \cdot g$

Input: $f = (f_0 + f_3 s) + (f_1 + f_4 s)t + (f_2 + f_5 s)t^2$ and $g = (g_0 + g_3 s) + g_1 t \in \mathbb{F}_{((p^2)^2)^3}$ with $f_j, g_0, g_1, g_3 \in \mathbb{F}_{p^2}$, $0 \le j \le 5$.

Output: $f \cdot g$.

```
1. $v_0 \leftarrow (g_0 + g_3 s)(f_0 + f_3 s)$, $v_1 \leftarrow g_1 \cdot (f_1 + f_4 s)$,
2. $u_0 \leftarrow g_1 \cdot ((f_1 + f_2) + (f_4 + f_5 s)$,
    $u_1 \leftarrow ((g_0 + g_1) + g_3 s)((f_0 + f_1) + (f_3 + f_4 s))$,
    $u_2 \leftarrow (g_0 + g_3 s)((f_0 + f_2) + (f_3 + f_5 s))$;
3. $c_0 \leftarrow v_0 + \xi(u_0 - v_1)$, $c_1 \leftarrow u_1 - v_0 - v_1$, $c_2 \leftarrow u_2 - v_0 + v_1$;
4. return $c_0 + c_1 t + c_2 t^2$;
```

Algorithm 4. Squaring of f in $\mathbb{G}_{\phi_{12}(p)}$ [16]

Input: $f = (f_0 + f_3 s) + (f_1 + f_4 s)t + (f_2 + f_5 s)t^2 \in \mathbb{F}_{((p^2)^2)^3}$ with $f_j \in \mathbb{F}_{p^2}$, $0 \le j \le 5$.

Output: f^2.

```
1. $v_0 \leftarrow f_0 f_3$, $v_1 \leftarrow f_1 f_4$, $v_2 \leftarrow f_2 f_5$, $A_0 \leftarrow f_0 + f_3$, $A_1 \leftarrow f_0 + \xi f_3$,
   $B_0 \leftarrow f_1 + f_4$, $B_1 \leftarrow f_1 + \xi f_4$, $C_0 \leftarrow f_2 + f_5$, $C_1 \leftarrow f_2 + \xi f_5$;
2. $u_0 \leftarrow A_0 A_1$, $u_1 \leftarrow B_0 B_1$, $u_2 \leftarrow C_0 C_1$,
    $A_0 \leftarrow v_0 + \xi v_0$, $B_0 \leftarrow v_1 + \xi v_1$, $C_0 \leftarrow v_2 + \xi v_2$;
3. $c_0 \leftarrow 3(u_0 - A_0) - 2f_0$, $c_1 \leftarrow 6v_2 + 2f_1$, $c_2 \leftarrow 3(u_1 - B_0) - 2f_2$,
   $c_3 \leftarrow 6v_0 + 2f_3$, $c_4 \leftarrow 3(u_2 - C_0) - 2f_4$, $c_5 \leftarrow 6v_1 + 2f_5$;
4. return $(c_0 + c_3 s) + (c_1 + c_4 s)t + (c_2 + c_5 s)t^2$;
```

Algorithm 5. Doubling step and f^2

```
Input: $f = a_0 + a_1 t + a_2 t^2$ with $a_0$, $a_1$, and $a_2 \in \mathbb{F}_{p^4}$; $P = (x_P, y_P) \in E(\mathbb{F}_p)$;
       $T = (X_T\tau^2, Y_T\tau^3, Z_T) \in E(\mathbb{F}_{p^{12}})$ with $X_T$, $Y_T$, and $Z_T \in \mathbb{F}_{p^2}$.
Output: $2T$, $l_{(T,T)}(P)$ and $f^2$.
1. $B \leftarrow Y_T^2$, $E \leftarrow 2Y_TZ_T$, $C \leftarrow 3Z_T^2$, $D \leftarrow 2X_TY_T$;
   $T_0 \leftarrow a_{0,0} + a_{0,1}$, $T_1 \leftarrow a_{0,0} + \xi a_{0,1}$;
2. $A \leftarrow X_T^2$, $U_0 \leftarrow a_{0,0}a_{0,1}$, $U_1 \leftarrow T_0T_1$;
   $g_3 \leftarrow B + iC$, $H \leftarrow 3C$, $F \leftarrow B + iH$, $G \leftarrow B - iH$;
   $T_0 \leftarrow a_{1,0} + a_{1,1}$, $T_1 \leftarrow a_{1,0} + \xi a_{1,1}$ ;
3. $g_0 \leftarrow Ey_P$, $J \leftarrow 4HC$, $g_1 \leftarrow -3Ax_P$, $I \leftarrow G^2$;
   $V_{0,0} \leftarrow U_1 - U_0 - \xi U_0$, $V_{0,1} \leftarrow 2U_0$ ;
4. $Z_{2T} \leftarrow 4BE$, $X_{2T} \leftarrow DF$, $U_0 \leftarrow a_{1,0}a_{1,1}$;
   $Y_{2T} \leftarrow I + J$, $A_0 \leftarrow a_{1,0} + a_{2,0}$, $A_1 \leftarrow a_{1,1} + a_{2,1}$;
5. $U_1 \leftarrow T_0T_1$, $W_0 \leftarrow a_{2,0}a_{2,1}$, $Z_0 \leftarrow A_0A_1$;
   $T_0 \leftarrow a_{2,0} + a_{2,1}$, $T_1 \leftarrow a_{2,0} + \xi a_{2,1}$, $X_0 \leftarrow A_0 + A_1$, $X_1 \leftarrow A_0 + \xi A_1$;
   $A_0 \leftarrow a_{0,0} + a_{1,0}$, $A_1 \leftarrow a_{0,1} + a_{1,1}$;
6. $W_1 \leftarrow T_0T_1$, $Z_1 \leftarrow X_0X_1$, $Y_0 \leftarrow A_0A_1$;
   $V_{1,0} \leftarrow U_1 - U_0 - \xi U_0$, $V_{1,1} \leftarrow 2U_0$, $X_0 \leftarrow A_0 + A_1$, $X_1 \leftarrow A_0 + \xi A_1$;
   $A_0 \leftarrow a_{0,0} + a_{2,0}$, $A_1 \leftarrow a_{0,1} + a_{2,1}$, $T_0 \leftarrow A_0 + A_1$, $T_1 \leftarrow A_0 + \xi A_1$;
7. $Y_1 \leftarrow X_0X_1$, $W_0 \leftarrow A_0A_1$, $W_1 \leftarrow T_0T_1$;
   $V_{2,0} \leftarrow W_1 - W_0 - \xi W_0$, $V_{2,1} \leftarrow 2W_0$, $V_{3,0} \leftarrow Z_1 - Z_0 - \xi Z_0$, $V_{3,1} \leftarrow 2Z_0$;
   $V_{3,0} \leftarrow V_{3,0} - V_{1,0} - V_{2,0}$, $V_{3,1} \leftarrow V_{3,1} - V_{1,1} - V_{2,1}$;
8. $c_{0,0} \leftarrow V_{0,0} + \xi V_{3,1}$, $c_{0,1} \leftarrow V_{0,1} + V_{3,0}$, $V_{3,0} \leftarrow Y_1 - Y_0 - \xi Y_0$, $V_{3,1} \leftarrow 2Y_0$;
   $c_{1,0} \leftarrow V_{3,0} - V_{0,0} - V_{1,0} + \xi V_{2,1}$, $c_{1,1} \leftarrow V_{3,1} - V_{0,1} - V_{1,1} + V_{2,0}$;
   $T_0 \leftarrow W_1 - W_0 - \xi W_0$, $T_1 \leftarrow 2W_0$;
   $c_{2,0} \leftarrow T_0 - V_{0,0} + V_{1,0} - V_{2,0}$, $c_{2,1} \leftarrow T_1 - V_{0,1} + V_{1,1} - V_{2,1}$;
9. return $(X_{2T}\tau^2, Y_{2T}\tau^3, Z_{2T})$, $g_0 + g_1\tau + g_3\tau^3$, $c_0 + c_1 t + c_2 t^2$;
```

Algorithm 6. Addition step

```
Input: $P = (x_P, y_P) \in E(\mathbb{F}_p)$, $Q = (x_Q\tau^2, y_Q\tau^3) \in E(\mathbb{F}_{p^{12}})$ and
       $T = (X_T\tau^2, Y_T\tau^3, Z_T) \in E(\mathbb{F}_{p^{12}})$ with $x_Q$, $y_Q$, $X_T$, $Y_T$, and $Z_T \in \mathbb{F}_{p^2}$.
Output: $T + Q$ and $l_{(T,Q)}(P)$.
1. $E \leftarrow x_QZ_T - X_T$, $F \leftarrow y_QZ_T - Y_T$;
2. $E_2 \leftarrow E^2$, $F_2 \leftarrow F^2$, $g_3 \leftarrow x_QF - y_QE$ ;
3. $B \leftarrow X_TE_2$, $E_3 \leftarrow EE_2$, $A \leftarrow Z_TF_2 - 2B - E_3$ ;
4. $X_{T+Q} \leftarrow AE$, $Z_{T+Q} \leftarrow Z_TE_3$, $g_0 \leftarrow Ey_P$, $g_1 \leftarrow -Fx_P$;
5. $Y_{T+Q} \leftarrow F(B - A) - y_QE_3$ ;
6. return $(X_{T+Q}\tau^2, Y_{T+Q}\tau^3, Z_{T+Q})$, $g_0 + g_1\tau + g_3\tau^3$;
```