Accelerating Performance of Bilinear Map Cryptography using FPGA

Andrei Ouatu
andrei.ouatu@upb.ro
Faculty of Computer Science
University Politehnica of Bucharest
Romania

Gabriel Ghinita
gghinita@hbku.edu.qa
College of Science and Engineering
Hamad Bin Khalifa University
Doha, Qatar

Razvan Rughinis
razvan.rughinis@upb.ro
Faculty of Computer Science
University Politehnica of Bucharest
Romania

ABSTRACT

Bilinear maps are used as an essential cryptographic building block in many of the advanced encryption algorithms today, such as searchable encryption, identity-based encryption, group signatures, etc. Numerous data and application privacy techniques make use of such primitives. However, the performance overhead of bilinear map encryption, and in particular that of the *pairing* operation, which is the predominant operation on bilinear maps, is still quite high. In this paper, we investigate in-depth the sequence of steps required to compute bilinear map pairings, and we identify the performance footprint of each step. We devise an implementation based on FPGA which reduces the overhead of bilinear pairings, in terms of both execution time and resource utilization (i.e., lookup tables and flip-flop units required). Our extensive performance evaluation shows that the proposed approach significantly outperforms benchmarks, and represents an important step towards the wide-scale deployment of bilinear map-based encryption protocol for large-scale applications.

CCS CONCEPTS

• **Security and privacy** → **Public key encryption**; • **Hardware** → **PCB design and layout**.

KEYWORDS

Bilinear Maps, FPGA

ACM Reference Format:
Andrei Ouatu, Gabriel Ghinita, and Razvan Rughinis. 2024. Accelerating Performance of Bilinear Map Cryptography using FPGA. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY '24), June 19–21, 2024, Porto, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3626232.3653250

1 INTRODUCTION

The last two decades witnessed an impressive number of novel cryptographic building blocks supporting advanced functionality [4, 7, 10] such as search on encrypted data, identity-based encryption, group signatures, etc. Many of these techniques make use of bilinear maps, a mathematical construction that has been popularized by the seminal work of Boneh et al [4]. Typically, a symmetrical bilinear map of composite order is employed, which is a function $e : \mathbb{G}\times\mathbb{G} \rightarrow \mathbb{G}_T$ such that $\forall a, b \in G$ and $\forall u, v \in \mathbb{Z}$ it holds that $e(a^u, b^v) = e(a, b)^{uv}$. $\mathbb{G}$ and $\mathbb{G}_T$ are cyclic multiplicative groups of composite order $N = P \cdot Q$ where P and Q are large primes of equal bit length. We denote by $\mathbb{G}_p$, $\mathbb{G}_q$ the subgroups of $\mathbb{G}$ of orders P and Q, respectively.

While bilinear maps are a very promising direction many application domains that rely on advanced encryption primitives, there are still barriers to large-scale adoption due to the significant performance overhead incurred. The fundamental operation when using bilinear maps is called *bilinear pairing*, and a single such operation can incur significant latency, in the order of milliseconds. For large datasets consisting of millions of data objects, the cumulative overhead can become prohibitive from a computational perspective.

Several research efforts focused on reducing the computational overhead of bilinear pairings, using either a software approach [1] or hardware-accelerated solutions [2, 3, 5, 6, 8, 9, 11]. Hardware approaches typically perform much faster than software approaches, due to the fact that bilinear maps are defined over fields of large bit-length, and general-purpose CPUs are not able to efficiently handle long word operations, especially when modular arithmetic on prime or composite fields is required. The advances in FPGA technology from the last decade, particularly in the area of specialized hardware for cryptographic acceleration, stimulated the development of several FPGA-based solutions for acceleration of bilinear pairings. However, existing approaches still have limitations in terms of latency and resource utilization on the FPGA board, leading to either approaches that are not sufficiently fast, or which do not scale well in terms of large-scale deployment and parallelism due to excessive hardware footprint.

The two essential metrics to quantify FPGA hardware resources are the number of required *lookup tables (LUT)* and *flip-flop units (FF)*. In this paper, we explore in-depth several design choices to efficiently implement bilinear pairings in hardware on FPGA devices. We focus on obtaining a good trade-off between low latency and reduced hardware footprint. We compare our approach with several state-of-the-art hardware and software implementations, and we show that we significantly outperform the benchmarks. The rest of the paper is organized as follows: Section 2 provides background information. We introduce our proposed approach for basic operations like modular multiplications and inversions in Sections 3 and 4 respectively, followed by an end-to-end illustration of the complete bilinear pairing pipeline in Section 5. We present experimental results in Section 6. We briefly overview related work in Section 7 and conclude in Section 8.

CODASPY '24, June 19–21, 2024, Porto, Portugal

ACM ISBN 979-8-4007-0421-5/24/06.
https://doi.org/10.1145/3626232.3653250

2 BACKGROUND

Our objective is to devise a hardware design to compute efficiently elliptic curve pairing operations on bilinear maps. Specifically, we focus on the Tate pairing [8, 9] which is a popular building block used in cryptographic constructions such as searchable encryption, identity-based encryption, group signatures, etc. The Tate pairing is defined over a super-singular elliptic curve described by equation:

$$E : y^2 + y = x^3 + x + 1 \tag{1}$$

where the curve parameters belong to a binary Galois Field GF(2^m), and the value of m is typically set to 283. Each Galois Field is defined by a finite field polynomial with binary coefficients

$$p(z) = \sum_{i=0}^{m-1} p_i z^i. \tag{2}$$

The polynomial can also be represented as a large integer

$$p(z) = p_{m-1} p_{m-2} \cdots p_0, \tag{3}$$

where p_i is the bit representing the coefficient of z^i.

Computing Tate pairings requires the implementation of polynomial arithmetic operations in GF(2^m). Specifically, as detailed in [8, 9], one needs to implement modular addition, modular squaring and modular multiplication in GF(2^m), as well as modular multiplication, modular exponentiation and modular inversion in GF(2^{4*m}).

Among the Galois Field arithmetic operations required to compute Tate pairings, we identify a trivial subset which can be easily and efficiently implemented using existing algorithms, as well as a more challenging subset that requires significant design efforts and optimizations to obtain good performance.

Specifically, the set of trivial operations consists of the following:

- Modular Addition in GF(2^m),
- Modular Squaring in GF(2^m),
- Modular Exponentiation in GF(2^{4*m}).

The set of challenging operations that represent the focus of our work, and for which we design customized hardware units, consists of the following:

- Modular Multiplication in GF(2^m) and GF(2^{4*m}),
- Modular Inversion in GF(2^m) and GF(2^{4*m}).

In the rest of this section, we provide a brief overview of the operations in the trivial set, and we also identify existing solutions from the literature that efficiently implement them. We present the details of our proposed solution for modular multiplication in Section 3, and for modular inversion in Section 4. In Section 5, we present the complete algorithm for Tate pairing computation, which includes both trivial and non-trivial operations.

2.1 Modular Addition in GF(2^m)

Starting from Eq. (2), it can be observed that the addition of two polynomials in GF(2^m) can be reduced to a single XOR operation between the binary representations of the two polynomials, where the i^{th} bit of the result is determined by computing the XOR between the i^{th} bits of the operand polynomials.

2.2 Modular Squaring in GF(2^m)

As detailed in [5], the squaring operation of a GF polynomial consists of expanding the binary representation of the polynomial from m-bits to $2*m$-bits, and inserting zeros at odd indices. Furthermore, the i^{th} index bit is moved to the $(2 \times i)^{th}$ index in the result. This approach also requires a reduction step to bring the result back to GF(2^m).

Given the reduction polynomial commonly used for Tate pairings $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$, the coefficients of the squaring result can be recombined using XOR gates after a factorization step for every power greater than 283. Each term in the result with an exponent exp greater than 283 is treated as $283 * (exp - 283)$ and x^{283} is replaced with $x^{12} + x^7 + x^5 + 1$.

2.3 Modular Exponentiation in GF(2^{4*m})

The modular exponentiation in GF(2^{4*m}) is required in the final step of the Tate Pairing algorithm (see Section 5 for details) and it is based upon the equivalences derived in [9, 11]. Specifically, by using the Frobenius map, the exponentiation of $c(x) = c_3*x^3+c_2*x^2+c_1*x+c_0 \in$ GF(2^{4*m}) defined over polynomial $f(x) = x^3 + x^2 + x + 1$ can be reduced to a series of XOR operations and recombinations of coefficients. Thus, the resulting value of $c(x)^{2^m}$ is given by the following equation:

$$c(x)^{2^m} = c_3 * x^3 + c_1 * x^2 + (c_2 + c_3) * x + (c_0 + c_1) \tag{4}$$

3 MODULAR MULTIPLICATION IN GALOIS FIELDS

We design specialized hardware units for modular multiplication operations in GF(2^m) and GF(2^{4*m}), and we encapsulate them in a separate Chisel3 [3] package imported in the hardware module that implements the Tate pairing algorithm. The Galois Field arithmetic operations are implemented starting from the multiplication algorithms presented in [5, 8].

3.1 Modular Multiplication in GF(2^m)

The hardware unit for GF(2^m) multiplication is the most important module in our design, and it is used as a building block in the other algorithms for multiplication in GF(2^m) and modular inversion. As a result, the GF(2^m) multiplication module directly impacts overall pairing performance, so it is essential to devise a hardware unit that achieves low latency and reduced resource consumption.

To highlight the challenges of efficiently implementing GF(2^m) multiplication, and to capture the benefit of each design choice in our solution, we present a sequence of candidate implementations, each one improving incrementally on the limitations of its predecessor. Table 1 summarizes each of these candidates, and briefly highlights its characteristics.

Algorithm M_0 (Baseline). The first implementation, referred to as M_0, or Baseline, follows the Algorithm 2.33 in [5]. As seen in Alg. 1, in this first naive implementation the computation of the result takes a number of cycles equal to m, the bit width of the Galois field elements. This is due to the fact that at the i^{th} iteration of the algorithm, the product $a_i * b(z)$ is XOR'd into the intermediate result. Even if the number of clock cycles needed for this approach is high, it still presents a series of advantages. First, the hardware

Table 1: Candidate Implementations of Modular Multiplication in GF(2^m)

Identifier	Method Description
M_0	This is a baseline approach that directly implements the algorithm from [5]. Each bit of the result is computed one-at-a-time in one clock cycle using in-place reduction.
M_1	This approach computes batches of the results in groups of ten bits at a time in a single clock cycle using in-place reduction.
M_2	This approach uses a digital serial multiplier which computes in a single clock cycle the product of two 32-bit words from the two operands and uses a separate reduction module.
M_3	This approach uses a digit-serial multiplier that divides the first operand into three (almost) equal-sized chunks, and multiplies each of these chunks with a 32-bit word from the second operand; the entire operation requires four different clock cycles.
M_4	This approach uses the same concept as M_3 but takes advantage of pre-computation techniques to reduce the total number of cycles.
M_5	In this approach, a digit-serial multiplier divides the first operand into four (almost) equal chunks and multiplies these chunks with a 32-bit word from the second operand, using four clock cycles. It also takes advantage of the pre-computation techniques to reduce the total number of cycles.

implementation of this approach is simple, and the resulting circuit consumes a small number of hardware resources. Second, the implementation executes the reduction step in-place, using only a series of XOR operations that can be derived from the rules of polynomial division. Thus, if the most significant bit of the binary representation of $b(z)$ (i.e., b_{m-1}) is set to one in the i^{th} iteration of the algorithm, the reduction can be done by performing a XOR of the coefficients for z^{12}, z^7 and z^5, and by setting the least significant bit to b_{m-1}. However, these advantages come at the cost of increased latency. As explained above, the total number of clock cycles needed for the entire computation is m, and one extra clock cycle is required for performing the instantiations.

Algorithm 1 M_0 - Baseline multiplication

Require: $poly1[m-1:0]$, $poly2[m-1:0]$
Ensure: $result[m-1:0] = poly1[m-1:0] * poly2[m-1:0])$
  $iter \leftarrow 0$;
  while $iter < m$ **do**
    if $iter = 0$ **then**
      $result \leftarrow poly1[0] * poly2$
    else
      $poly2 \leftarrow poly2 << 1$
      $poly2 \leftarrow poly2 \oplus (poly2[m-1] * f(x))$
      $result \leftarrow result \oplus (poly1[iter] * poly2)$
    end if
    $iter \leftarrow iter + 1$
  end while

Algorithm M_1 (Operand Bit Batching). Starting from the high-latency implementation of the baseline approach, we devise a second implementation, referred to as M_1. Instead of multiplying only one bit from the first operand with the second operand in the same clock cycle, ten bits of the first operand are multiplied with the second operand. This approach reduces the number of clock cycles needed for the entire computation from the bitwidth of the operands (m) to $m/10$. However, the manipulation of multiple bits in each clock cycle increases the complexity of the circuit. Equation (5) illustrates how the multiplication is performed in the i^{th} iteration of M_1:

$$a_{\mathrm{i}}(z) * b(z) = a_{\mathrm{j}} * b(z) + a_{\mathrm{j+1}} * b(z) + \ldots + a_{\mathrm{j+9}} * b(z), \quad (5)$$

where a_{i}(z) describes the chunk from the first operand equal to $a_{\mathrm{i}^*10} * z^{i*10} + a_{\mathrm{i}^*10+1} * z^{i*10+1} + \ldots + a_{\mathrm{i}^*10+9} * z^{i*10+9}$ and $j = i * 10$. When translated to hardware, this computation consists of XOR operations between a series of multiplexers (MUX). The MUXs are tasked to manage the multiplication between a_j and $b(z)$. Thereby, based on the value of a_j as selection signal, each MUX unit returns either 0 or $b(z) * z^j$. The computation of $b(z) * z^j$ consists of left shifting the binary representation of $b(z)$ by j positions. Next, this intermediate result needs to be reduced to the Galois field space. As in the previous implementation of M_0, the reduction step is done in-place. In this manner, all the powers of the intermediate result that are greater than the width of the Galois field can be decomposed into $z^{283} * z^{power-283}$. Using this factorization and the fact that z^{283} can be written as $z^{12} + z^7 + z^5 + 1$, the reduction step consists of only a series of XOR gates and recombination of the coefficients. Even if this approach reduces the latency for the multiplication in the GF(2^m) hardware unit, it requires ten left shift operations in the same clock cycle together with the usage of MUXs to check if the shifts are necessary. It also requires the use of XOR gates in the coefficients' recombination in the reduction step, thus further increasing the circuit complexity and the number of hardware resources consumed by this approach. Although faster, M_1 consumes three times more hardware resources than M_0.

Algorithm M_2 (Digit-Serial Multiplier). The third candidate implementation for Galois field modular multiplication builds upon the digit-serial multiplier approach introduced in [5, 8].

The digit-serial multiplier increases the size of the chunk from the second operand that is processed in a single clock cycle, thus improving the latency of the entire circuit. Specifically, a 32-bit

Algorithm 2 M_1: Operand Bit Batching

Require: $poly1[m-1:0], poly2[m-1:0], f(x)$
Ensure: $result[m-1:0] = poly1[m-1:0] * poly2[m-1:0])$
  $iter \leftarrow 0$;
  while $iter < m$ **do**
    if $iter = 0$ **then**
      $result \leftarrow poly1[0] * poly2$
      $iter \leftarrow iter + 1$
    else
      $poly2 \leftarrow poly2 * poly1[iter + 9 : iter]$
      $poly2 \leftarrow poly2 \mod f(x)$
      $result \leftarrow result \oplus poly2$
      $iter \leftarrow iter + 10$
    end if
  end while

Algorithm 3 M_2: Digital Serial Multiplier

Require: $poly1[m-1:0], poly2[m-1:0], f(x), wordSize$
Ensure: $result[m-1:0] = poly1[m-1:0] * poly2[m-1:0])$
  for $i \leftarrow 0$ to $\lceil \frac{m}{wordSize} \rceil$ **do**
    $result \leftarrow result \oplus (poly1 * poly2[(i+1) * wordSize - 1 : i * wordSize])$
    $poly1 \leftarrow (poly1 << wordSize) \mod f(x)$
  end for
  $result \leftarrow result \mod f(x)$

word from the second operand is multiplied with the first operand in each clock cycle. As a result, the number of clock cycles is reduced to only $\lceil \frac{width}{32} \rceil$ if the multiplication of the word $B_i(z)$ with $a(z)$ can be done in a single clock cycle. Compared to the two previous approaches, the use of the digit-serial multiplier enforces the necessity to implement a separate reduction module that follows NIST standards for GF(2^{283}) with $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$ as the reduction polynomial. The reduction module follows the pseudocode given as Algorithm 2 in [9] and Algorithm 2.43 in [5]. Thus, by following the reduction algorithm, we are able to reduce a 566-bit polynomial to a 283-bit polynomial in GF(2^{283}), requiring a single clock cycle for processing each 32-bit word that contains the coefficients for the powers greater than 283 in the polynomial to be reduced. As illustrated in Algorithm 3, the polynomial that we need to reduce back to GF(2^{283}) has at most 315 bits, because we multiply a 32-bit operand with a 283-bit operand. The reduction module can be reduced to only one clock cycle compared to the reduction module proposed in [8], which takes four clock cycles to reduce the polynomial resulting after multiplication. Compared to the implementation from [8], our reduction module processes only the word containing the powers up to 313, thus decreasing the total latency brought by the reduction module, and also the hardware resources consumed by our design, because it can be implemented as a combinational logic circuit. Algorithm 4 illustrates the reduction module implemented in our work to be used within the GF(2^m) modular multiplication hardware unit.

Algorithm 4 Algorithm for polynomial reduction

Require: $poly[m-1:0], f(x), wordSize$
  $t \leftarrow poly[10 * wordSize - 1 : 9 * wordSize]$
  $c_0 \leftarrow poly \oplus (t << 5) \oplus (t << 10) \oplus (t << 12) \oplus (t << 17)$
  $c_1 \leftarrow poly \oplus (t >> 27) \oplus (t >> 10) \oplus (t >> 12) \oplus (t >> 17)$
  $t \leftarrow (poly[9 * wordSize - 1 : 8 * wordSize] >> 27)$
  $result \leftarrow (t\&0x7FFFFF, poly[8 * wordsize - 1 : 2 * wordsize], c_1, c_0 \oplus t \oplus (t << 5) \oplus (t << 7) \oplus (t << 12))$

By using the digit-serial multiplier approach, M_2 requires a single clock cycle to compute the coefficient for each power of the intermediate result, thus decreasing the latency of the bilinear pairing computation, However, it introduces an increase in the number of hardware resources consumed, caused by the calculation of all the coefficients in the same clock cycle, as illustrated in Algorithm 3.

Algorithms M_3, M_4 (Digit-Serial Multiplier with Batching). To address the increased hardware resource consumption of M_2, we propose an improved approach called M_3. Illustrated in Algorithm 5, M_3 performs multiplication between $B_i(z)$ and $a(z)$ in four clock cycles only. It implements a finite state machine with four different states, whereby $B_i(z)$ is multiplied with a different chunk of a(z) with size equal to $\lceil \frac{m}{3} \rceil$ in the first three states, while the last state outputs the final result. The multiplication in each different state of this approach is done by using the rules of polynomial multiplication, and by taking into account which powers of the result are affected at each step. This way, we obtain a significant reduction in the number of LUTs (Look-Up Tables) required for the entire system. However, the latency of the entire system is slightly increased, due to the introduction of additional states in the computation of $B_i(z) * a(z)$. The total amount of clock cycles required for the entire modular multiplication in GF(2^m) is similar to the one in [8].

Algorithm 5 M_3 and M_4 Algorithms

Require: $poly1[m-1:0], poly2[m-1:0], f(x), wordSize$
Ensure: $result[m-1:0] = poly1[m-1:0] * poly2[m-1:0])$
  Separate $poly$ into three equal chunks
  $aux\_res[m + wordSize - 1 : 0] \leftarrow 0$
  for $i \leftarrow 0$ to $\lceil \frac{m}{wordSize} \rceil$ **do**
    1. $aux\_res \leftarrow chunk1 * poly2$
    2. $aux\_res \leftarrow chunk2 * poly2$
    3. $aux\_res \leftarrow chunk3 * poly2$
    {Multiplications 1-3 are sequential}
    $result \leftarrow result \oplus aux\_res$
    $poly1 \leftarrow (poly1 << wordSize) \mod f(x)$
  end for
  $result \leftarrow result \mod f(x)$

Algorithm M_4 operates the same way as M_3, with the only difference that M_4 employs pre-computation techniques described later in Algorithm 10.

Algorithm M_5 (Digit-Serial Multiplier with fine-grained Batching). The last variant we consider, referred to as M_5, implements the multiplication $B_i(z) * a(z)$ using a finite state machine with *six* states, and separates $a(z)$ into four chunks of size equal to $\lceil \frac{m}{4} \rceil$.

The approach is illustrated in Algorithm 6. The first four states implement the multiplication of each equal chunk of a(z) with $B_i(z)$, while the fifth state multiplies the most significant bits of $a(z)$ that were not part of any of the initial chunks with $B_i(z)$. The final state outputs the result together with an additional signal *outputValid* which indicates that the output of the module is the final result. Even though this approach increases slightly the latency of the entire system, it reduces the number of LUTs required by our design, while the number of Flip-Flop units (FF) remains similar to the previous variants M_3 and M_4. We found that this implementation has the best overall performance when combined with the pre-computing techniques described in Algorithm 10.

Algorithm 6 Algorithm for M_5

Require: $poly1[m-1:0], poly2[m-1:0], f(x), wordSize$
Ensure: $result[m-1:0] = poly1[m-1:0] * poly2[m-1:0])$
  Separate $poly1$ into four equal chunks
  $aux\_res[m + wordSize - 1:0] \leftarrow 0$
  for $i \leftarrow 0$ to $\lceil \frac{m}{wordSize} \rceil$ **do**
    1. $aux\_res \leftarrow chunk1 * poly2$
    2. $aux\_res \leftarrow chunk2 * poly2$
    3. $aux\_res \leftarrow chunk3 * poly2$
    4. $aux\_res \leftarrow chunk4 * poly2$
    {The multiplications between the chunks and poly2 are done sequentially}
    $result \leftarrow result \oplus aux\_res$
    $poly1 \leftarrow (poly1 << wordSize) \mod f(x)$
  end for
  $result \leftarrow result \mod f(x)$

3.2 Modular Multiplication in GF(2^{4*m})

Our proposed implementation for this hardware unit follows the Karatsuba multiplier idea proposed in [6] using the circuit schematic provided in [8] and depicted in Fig. 1.

Starting from this architecture of the GF(2^{4*m}) modular multiplier defined over $f(x) = x^3 + x^2 + x + 1$, we consider two distinct approaches. The first approach aims to improve the latency of the entire pairing operation by computing all of the multiplications in Fig. 1 in parallel, using nine different multipliers in GF(2^m), thus increasing the amount of hardware resources that the design consumes. The block diagram of this low-latency solution is illustrated in Fig. 2. The second approach aims to reduce the amount of LUTs and FFs utilized by the FPGA, at the expense of higher latency, by using only one multiplier in GF(2^m) reused for each of the multiplications shown in Fig. 1. Thus, the two approaches provide a trade-off between latency and resource consumption.

The low-latency implementation for the schematic in Fig. 1 uses a 2-state finite-state machine (FSM). The first state computes the XOR operations in Fig. 1 and the nine parallel multiplications. Then, based upon the validity signal that indicates whether the results of the multiplications can be used in further computations, the FSM moves to the second state where the final XOR operations are computed and the final result is output, marking the validity of the result with a signal. To store the results from the first state computations and reuse them in the second state, we use nine registers with a size equal to m. The use of the two-state FSM delays with one clock cycle the output of the final result. However, since the split in two different states is not mandatory, the circuit can be reduced to only one state in which all of the computations are performed, by taking advantage of the fact that XOR operations are independent of the clock. Using this observation, there is no need for the intermediate result registers, and hence the design is simplified, and the amount of FFs used by the system is significantly reduced. Latency is also decreased, as the validity signal from this module becomes true one clock cycle before the equivalent 2-state FSM would.

4 MODULAR INVERSION IN GALOIS FIELDS

4.1 Modular Inversion in GF(2^m)

Algorithm 7 Algorithm for Modular Inversion in GF(2^m)

Require: $polynomial[m-1:0]$
Ensure: $result[m-1:0] = polynomial^{-1}[m-1:0]$
  $a \leftarrow polynomial^2, b \leftarrow 1, x \leftarrow \frac{m-1}{2}$
  while $x \neq 0$ **do**
    $a \leftarrow a * a^{2^x}$
    if $x\%2$ **then**
      $b \leftarrow a * b$
      $a \leftarrow a^2$
      $x \leftarrow \frac{x-1}{2}$
    else
      $x \leftarrow \frac{x}{2}$
    end if
  end while
  $result \leftarrow b$

Our implementation of the inversion operation in GF(2^m) is illustrated in Algorithm 7 and Fig. 3. This hardware unit uses a Modular Multiplier in GF(2^m) and a Modular Squarer in GF(2^m) that are re-utilized during the different steps of the algorithm. To translate Algorithm 7 in hardware, we employ a 5-state FSM, as follows. The first state is an initialization step, whereas the second state is tasked with computing A^{2^x} by using the Modular Squarer in GF(2^m) integrated in this unit. The third state uses the Modular Multiplier in GF(2^m) to compute $A * A^{2^x}$. The fourth state updates either variable x or the result of $B * A$, depending on the parity of x. The final state is activated only if x is odd, and it updates the value of A with A^2. The implementation cycles through these states until x becomes 0, signaling that the computation is complete. For $m = 283$, a total of 283 squaring operations in GF(2^m) and 12 multiplications in GF(2^m) are required.

4.2 Modular Inverter in GF(2^{4*m})

The modular inversion in GF(2^{4*m}) needed in the final steps of the bilinear pairing operation is implemented by using a modular multiplier in GF(2^{4*m}), a modular inverter in GF(2^m) and a modular exponentiator in GF(2^{4*m}), as shown in Fig. 3. This hardware unit is the most complex among all Galois field operations implemented in our package, and also the most hardware-demanding, consuming the most LUTs and FFs. The implementation in our work follows

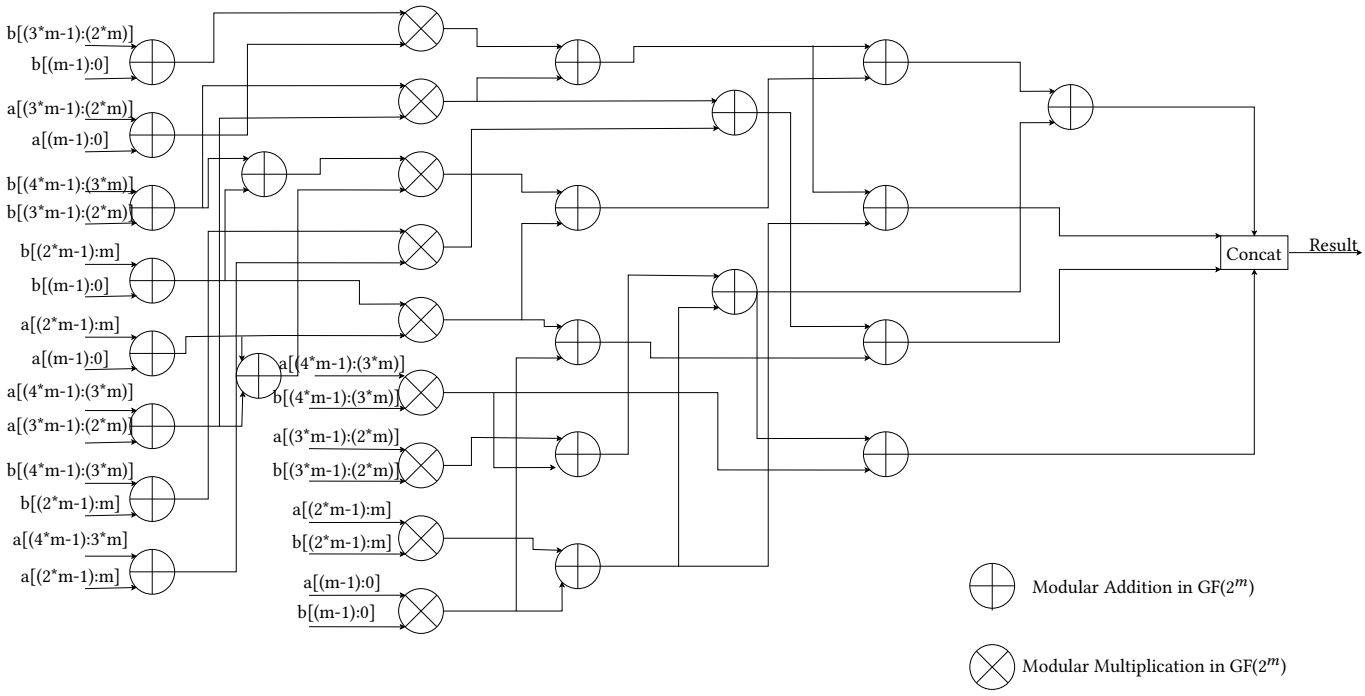

Figure 1: Data flow of Karatsuba Multiplier implemented for Modular Multiplication in GF(2^{4*m})

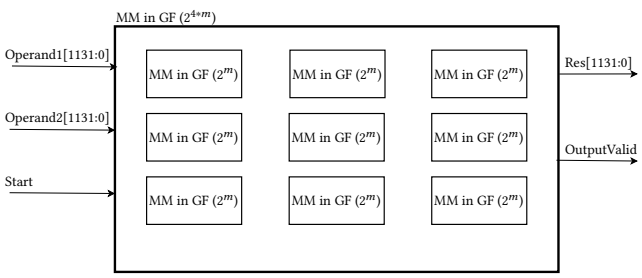

Figure 2: Block diagram of Modular Multiplier in GF(2^{4m})

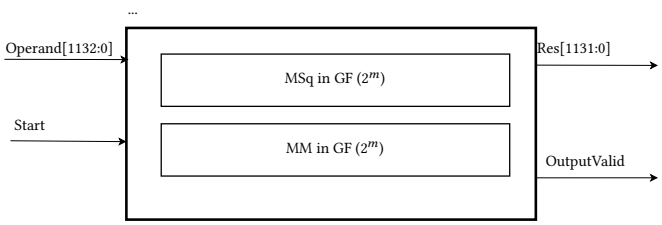

Figure 3: Block diagram of Modular Inverter in GF(2^m)

Algorithm 3 in [8], and relies on the following formulae derived in [8]:

$$p(x)^{-1} = p(x)^{2^{4*m}-2} = (p(x)^r)^{-1} * p(x)^{r-1}, \quad (6)$$

where $r = \frac{2^{4*m}-2}{2^m-1}$.

$$p(x)^{r-1} = ((p(x)^{2^m} * p(x))^{2^m} * p(x))^{2^m} \quad (7)$$

To implement the algorithm, we used a FSM with eight states. The first stage focuses on the computation of $p(x)^{2^{283}} * p(x)$ by using the modular exponentiator in GF(2^{4*m}) for $p(x)^{2^{283}}$ and the modular multiplier in GF(2^{4*m}). These two operations are performed in the same stage to save a clock cycle and hardware resources that would have been needed to propagate the results from one operation to another. By taking advantage of the nature of exponentiation in GF(2^{4*m}), we perform only a recombination of coefficients which we implement within a combinational circuit. The transition to the next state is done when the *outputValid* signal of the modular multiplier is set to true. The second state is used to reset the internals of the modular multiplier and takes only one clock cycle. The third state computes the value $(p(x)^{2^{283}} * p(x))^{2^{283}} * p(x)$ based on the result of $p(x)^{2^{283}} * p(x)$ which has already been computed in the first state and stored in a register. The fourth state of the FSM computes the final exponentiation from the formula above, resets the multiplier in GF(2^{4*m}) and stores the result in a register. The fifth state calculates $p(x)^r$ and transitions to the sixth state where the inversion in GF(2^m) is computed. The seventh state computes the final multiplication by using the results from the sixth state and the fourth state. It propagates the final result and sets the *outputValid* signal to true marking that the result can be used in further computations downstream.

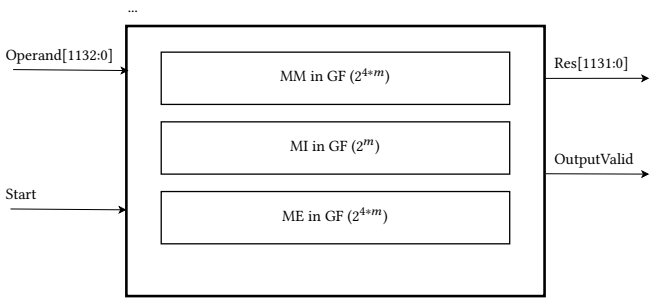

Figure 4: Block diagram of Modular Inverter in GF(2^{4m})

Algorithm 8 Algorithm for Modular Inversion in GF(2^{4*m})

Require: $polynomial[4*m-1:0]$
Ensure: $result[4*m-1:0] = polynomial^{-1}[4*m-1:0]$
$auxPoly \leftarrow polynomial^{2^m}$
$auxPoly \leftarrow auxPoly * polynomial$
$auxPoly \leftarrow auxPoly^{2^m}$
$auxPoly \leftarrow auxPoly * polynomial$
$auxPoly \leftarrow auxPoly^{2^m}$
$auxPoly2 \leftarrow auxPoly * polynomial$
$auxPoly2 \leftarrow auxPoly2^{-1}$
$result \leftarrow auxPoly * auxPoly2$

5 COMPLETE BILINEAR PAIRING ALGORITHM

In this section, we provide the complete algorithm for bilinear pairing implementation, which combines all techniques proposed in the previous sections. We use a modular multiplier in GF(2^{4*m}), a modular exponentiator in GF(2^{4*m}), a modular inverter in GF(2^{4*m}), two modular squarers in GF(2^m) and six modular multipliers in GF(2^m).

The algorithm shown in Algorithm 9 is divided into two sections. First section includes the initialization step and the loop step, represented by the first two states of the diagram in Fig. 5. The second section includes the calculation of $C(x)^{2^{2*m}-1}$ and consists of the final two states in Fig. 5. For the first section we use a FSM with eight different states. In the second section, the computation of $C(x)^{2^{2*m}-1}$ is translated to $C(x)^{2^{2*m}} * C(x)^{-1}$.

The states of the FSM tasked with the first section of Algorithm 9 are presented in Fig. 6. The first state of the FSM is used for the initialization step, taking only one clock cycle and moving the FSM to its second state. In the second state x_p and y_p are squared using the modular squarers in GF(2^m) taking one clock cycle. Then the FSM moves to its third state where z and $m1$ are computed based upon on the values of x_p and y_p from the previous state. When the multiplication between x_p and y_p is complete, the FSM enters the fourth state, tasked with the computation of w and the reset of the multipliers that will be used in the next state. In the fifth state in which the values of m_2, m_3, m_4, m_5, m_6 and m_7 are computed there are two possible approaches: one oriented to reduced latency in which all of the multiplications needed are done in parallel and therefore six different multipliers in GF(2^m) are used; and

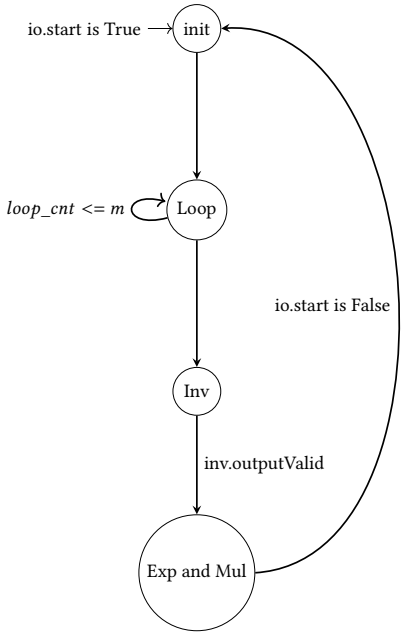

Figure 5: State diagram of the Tate Pairing Unit

a second approach oriented to reduce the hardware area of the design in which only one multiplier in GF(2^m) is used but all of the multiplications are done sequentially. We prefer the first approach due to its improved latency. When all the multiplications in this stage are completed, the FSM moves to its sixth state tasked with the calculation of the new values for c_0, c_1, c_2 and c_3 and with the reset of the multipliers. The transition to the seventh state is done after one clock cycle. The seventh state loops for *m - 1* times in order to compute ${x_q}^{2^{m-1}}$ and ${y_q}^{2^{m-1}}$ taking thus *m - 1* clock cycles and representing the bottleneck of this implementation. After *m - 1* clock cycles the FSM gets to its final state where the counter denoting the current step of the loop is incremented and the FSM moves back to the second state repeating this until the counter is greater than *m*. When this internal counter becomes greater than *m* the algorithm moves to the second section.

In the second section the computation of $C(x)^{2^{2*m}} * C(x)^{-1}$ is separated in two different stages as seen in Fig. 7. The first phase computes $C(x)^{-1}$ using the modular inverter in GF(2^{4*m}) while the second phase computes $C(x)^{2^{2*m}}$ using the modular exponentiator in GF(2^{4*m}) and then multiplies it with the result of the inversion from the previous phase using the modular multiplier in GF(2^{4*m}). When this multiplication is over the final result is output and the *outputValid* signal of the pairing unit is set to true.

As mentioned above, the bottleneck in terms of latency of this design is represented by the computation of ${x_q}^{2^{m-1}}$ and ${y_q}^{2^{m-1}}$ which takes *m - 1* clock cycles in each loop. A closer examination shows that these computations can be moved outside of the loop, which reduces the number of states of the FSM to just six, thus reducing the latency of the entire system. Algorithm 10 illustrates how this pre-computation step consists of calculating the values of x_q and y_q needed into each loop, and storing them into LUTs

before starting the computation of the bilinear pairing. By performing the pre-computation, the implementation reduces the latency considerably. To reduce hardware resource consumption due to pre-computation, we employ the M_5 version of the modular multiplier in GF(2^m).

Algorithm 9 Algorithm for Bilinear Pairing without pre-computation

Require: $P(x_p, y_p), Q(x_q, y_q), x_p, y_p, x_q, y_q \in GF(2^m)$
Ensure: $c(x) = e(P, Q), c(x) \in GF(2^{4*m})$
$c(x) = c_3 * x^3 + c_2 * x^2 + c_1 * x + c_0 = 1$
for $i \leftarrow 0$ to m **do**
$x_p \leftarrow {x_p}^2, y_p \leftarrow {y_p}^2$
$z \leftarrow x_p \oplus y_p, m_1 \leftarrow x_p * x_q$
$w \leftarrow z \oplus m1 \oplus y_p \oplus y_q \oplus 1$
$m_2 \leftarrow c_0 * w$,
$m_3 \leftarrow (c_2 \oplus c_3) * (z \oplus 1)$,
$m_4 \leftarrow (c_1 \oplus c_2 \oplus c_3) * w$,
$m_5 \leftarrow (c_0 \oplus c_2 \oplus c_3) * (w \oplus z \oplus 1)$,
$m_6 \leftarrow c_3 * (z \oplus 1)$,
$m_7 \leftarrow (c_1 \oplus c_2) * (w \oplus z \oplus 1)$
$c_0 \leftarrow m_2 \oplus m_3 \oplus c_3$,
$c_1 \leftarrow m_2 \oplus m_4 \oplus m_5 \oplus m_6 \oplus c_0 \oplus c_3$,
$c_2 \leftarrow m_2 \oplus m_4 \oplus m_5 \oplus m_7 \oplus c_1$,
$c_3 \leftarrow m_4 \oplus m_7 \oplus c_2$
$x_q \leftarrow {x_q}^{2^{m-1}}, y_q \leftarrow {y_q}^{2^{m-1}}$
end for
$c(x) \leftarrow c(x)^{2^{2*m}} * c(x)^{-1}$

Algorithm 10 Algorithm for Bilinear Pairing with pre-computation

Require: $P(x_p, y_p), Q(x_q, y_q), x_p, y_p, x_q, y_q \in GF(2^m)$
Ensure: $c(x) = e(P, Q), c(x) \in GF(2^{4*m})$
$c(x) = c_3 * x^3 + c_2 * x^2 + c_1 * x + c_0 = 1$
Load data for values calculated for ${x_q}^{2^{m-1}}$ and ${y_q}^{2^{m-1}}$ into arrays with size $m-1$ named x_q_values and y_q_values
for $i \leftarrow 1$ to m **do**
$x_p \leftarrow {x_p}^2, y_p \leftarrow {y_p}^2$
$z \leftarrow x_p \oplus y_p, m_1 \leftarrow x_p * x_q\_values[i-1]$
$w \leftarrow z \oplus m1 \oplus y_p \oplus y_q\_values[i-1] \oplus 1$
$m_2 \leftarrow c_0 * w$,
$m_3 \leftarrow (c_2 \oplus c_3) * (z \oplus 1)$,
$m_4 \leftarrow (c_1 \oplus c_2 \oplus c_3) * w$,
$m_5 \leftarrow (c_0 \oplus c_2 \oplus c_3) * (w \oplus z \oplus 1)$,
$m_6 \leftarrow c_3 * (z \oplus 1)$,
$m_7 \leftarrow (c_1 \oplus c_2) * (w \oplus z \oplus 1)$
$c_0 \leftarrow m_2 \oplus m_3 \oplus c_3$,
$c_1 \leftarrow m_2 \oplus m_4 \oplus m_5 \oplus m_6 \oplus c_0 \oplus c_3$,
$c_2 \leftarrow m_2 \oplus m_4 \oplus m_5 \oplus m_7 \oplus c_1$,
$c_3 \leftarrow m_4 \oplus m_7 \oplus c_2$
end for
$c(x) \leftarrow c(x)^{2^{2*m}} * c(x)^{-1}$

6 EXPERIMENTAL EVALUATION

All hardware units were synthesized using Vivado 2023.1, target board Virtex-7 VC709 Evaluation Platform XC7VX690TFFG1761-2. Algorithms were implemented using Chisel3 [3], and subsequently each unit was translated into Verilog using Scala *sbt*. The synthesis directive set in our evaluation was *LogicCompaction* while using a "full flatten" hierarchy and out-of-context mode. Table 2 shows the hardware area occupied by our modules after the synthesis step, while Table 3 compares our implementations for the modular arithmetic operations with the ones proposed in [8]. Table 4 shows the hardware area occupied by our complete bilinear pairing system (using the Tate pairing). The metrics used are number of LUTs (Look-Up Tables) and FFs (Flip-Flops) needed for each unit.

To measure the latency of our implementation, (i.e., the total time needed for the pairing computation) we use the Vivado 2023.1 Synthesis Report Timing Summary and the Verilator [2] simulator. The total latency is presented in Table 5, compared to the benchmark system proposed in [8].

Table 2: Performance of Modular Multiplication in GF(2^m)

Multiplier Variant	Hardware Resources	
	LUTs	FFs
M_0 - Baseline	1054	579
M_1	3268	292
M_2	5363	861
M_3, M_4	3200	871
M_5	2440	864

Table 3: All GF(2^m) operations vs Benchmark

Hardware Module	Hardware Resources			
	Our implementation		Li et al. [8]	
	LUTs	**FFs**	**LUTs**	**FFs**
Modular Squaring in GF(2^{283})	273	0	567	317
Modular Multiplier in GF(2^{283}) - M_5	2440	864	3367	2156
Modular Reduction in GF(2^{283})	54	0	1439	825
Modular Inversion in GF(2^{283})	3972	1723	13944	6999
Modular Multiplier in GF(2^{4*283})	23560	7776	48591	32578
Modular Inversion in GF(2^{4*283})	31820	13212	54988	43746

The M_0 baseline consumes the least amount of hardware resources compared to the other variants, both for a single multiplication operation (as shown in Table 2), as well as the complete pairing computation pipeline (Table 4). The main disadvantage of M_0 is increased latency, as shown in Table 5. When compared to the implementation from [8], the complete pairing operation using M_0 incurs a latency three times greater than competitors. The M_1 approach brings an improvement in terms of latency compared to M_0 but the hardware area occupied by the entire system increases considerably as a consequence of the left shifts needed and auxiliary MUX units used in the reduction in-place step. When using M_2, the latency decreases to half that of M_0, but the number of LUTs needed is approximately three times greater and the number of FFs increases by 15%. This increase in terms of hardware resources is caused by the computation of $B_i(z) * a$ in a single clock cycle. The other variants for modular multiplication in GF(2^m), M_3/M_4 and M_5,

bring an improvement in terms of hardware resources consumed as shown in Tables 2 and Table 4, mainly due to performing multiplication $B_i(z) * a$ in multiple clock cycles. As presented in Table 5, the complete bilinear pairing operation using M_3 shows an increase in latency because more clock cycles are needed to compute $B_i(z) * a$. However, in terms of hardware area, the number of LUTs used drops by approximately 39% and the number of FFs by approximately 15% compared. When compared to [8] the LUTs consumed are 7% less, while the FFs consumed are 47% less, but the latency is still higher. A pairing operation that uses M_4 or M_5 has a smaller latency compared to [8] as a consequence of the pre-computation mechanism that is used in these implementations, and the usage of the reduction module that needs only one clock cycle to reduce the polynomials resulted after multiplication back to GF(2^m). Among these two implementations, we prefer M_5 (shown bold in Table 4 and Table 5). This variant present a latency approximately three times smaller compared to [8] while the number of LUTs it uses is approximately 32%, and the number of FFs needed to operate correctly is approximately 47%.

Table 4: Performance of complete bilinear pairing

Bilinear Pairing Operation	Hardware Resources	
	LUTs	FFs
M_0 - Baseline	50380	30574
M_1	111815	23300
M_2	160052	35465
M_3	97605	30956
M_4	107985	30909
M_5	**71014**	**30742**
Li et al [8]	*104860*	*57753*

Table 3 compares our implementations in terms of hardware resources to the hardware modules proposed in [8]. Our hardware modules consume fewer resources mainly because our Modular Multiplier in GF(2^m) uses less LUTs and less FFs to operate correctly. This is possible due to the separation of $B_i(z) * a$ operation in multiple stages compared to [8] in which this operation is done in only one clock cycle. The smaller amount of LUTs and FFs consumed by the Modular Squaring in GF(2^m) unit makes the Modular Inversion in GF(2^m) and Modular Inversion in GF(2^{4*m}) less resource-consuming when compared to the implementation in [8]. Furthermore, our implementations do not incur any DSP (Digital Signal Processor) slices. This is due to the multiplication rules in binary Galois Fields. In GF(2^m) the multiplication is translated to AND operations between the bits from the binary sequences denoting the operands.

Finally, we compare our hardware approach with the existing state-of-the-art software approaches implemented in the MIRACL library [1]. The software implementation from the MIRACL library was run on a desktop computer with Windows Subsystem for Linux 2 on Windows 10, a 2.4 GHz Intel Core i5 9300H processor and 16 GB of memory, g++ compiler version 9.4.0. The speedup obtained by our implementations is presented in Table 6. The slowest of our implementations still yields a considerable 8.6 speedup over the software approach, while the M_4 method achieves a speedup of 127. The variant that uses M_5 and pre-computation, which has a low hardware footprint on the FPGA chip, achieves a significant 80.5 speedup over the software implementation in the MIRACL library, proving that the hardware approach using specialized FPGAs is a powerful method to compute bilinear pairings.

Table 5: Latency of the complete pairing operation

Variant	Latency (ms)
M_0 - Baseline	1.77
M_1	1.10
M_2	0.80
M_3	0.89
M_4	0.12
M_5	0.19
Li et al [8]	*0.59*

Table 6: Speedup over software benchmark

Variant	Speedup over software
M_0 - Baseline	8.6
M_1	13.9
M_2	19.1
M_3	17.2
M_4	127.5
M_5	80.5

Table 7: Maximum operating frequency

Variant	Maximum operating frequency (MHz)
M_0 - Baseline	172.56
M_1	151.37
M_2	189.61
M_3	182.31
M_4	173.31
M_5	161.66
Li et al [8]	*159.76*

7 RELATED WORK

Most existing approaches for computing bilinear map pairings start from implementing units specialized in Galois Fields arithmetic primitives. These include Modular Squaring, Modular Multiplication, Modular Exponentiation and Modular Inversion, which are used as building blocks in the bilinear map pairing unit design. The Modular Multiplication in GF(2^m) unit impacts the most the computation overhead of the pairing.

McCusker et al. [9] present an approach in which Galois Fields arithmetic is implemented in hardware using a bit-serial multiplication module. They achieve a reduction in terms of hardware resources consumed by multiplication, but at the cost of an increased latency of the circuit.

Hankerson et al. [5] show how Galois Finite Field arithmetic can be implemented in software and in hardware. Their book presents standard algorithms offered by NIST for the Modular Reduction needed in the context of Modular Multiplication. These algorithms are specific for the Galois Field width intended to be used. In terms of Modular Multiplication in GF(2^m) they present multiple possible

approaches starting from a bit-serial multiplier to a digit-serial multiplier.

Li et al. [8] implement a digit-serial multiplier alongside a fast modular reduction module. Their approach consists of separating one of the operands of the Modular Multiplication in GF(2^m) in multiple chunks with a size equal to 32. Together with that they design a Modular Squarer that does not increase in size with the operand, followed by a reduction step. Their squaring unit only needs a coefficients recombination step for the resulting polynomial based upon the characteristics of the reduction polynomial. The authors also implement the Tate pairing algorithm proposed in [9]. Their hardware synthesis shows that the design requires a considerable amount of hardware resources in terms of LUTs and FFs.

All of the solutions discussed above use the Karatsuba multiplier described in [6] to implement the Modular Multiplication in GF(2^{4*m}) while their Modular Exponentiation operation is based upon observations and proofs shown in [11].

8 CONCLUSIONS

We proposed a hardware unit for accelerated computation of bilinear map pairings on FPGAs which achieves a superior trade-off between latency on the one hand, and resource consumption in the form of LUT and FF on the other. Our proof-of-concept implementation shows significant improvements compared to benchmarks. In future work, we plan to investigate hardware designs that use multiple FPGAs to parallelize the execution of bilinear pairings and scale to massive datasets. We will also explore cross-operation computation reuse to further decrease latency and reduce resource consumption.

Acknowledgments. This research has been funded in part by NSF grant IIS-1909806 and an unrestricted cash gift from Microsoft Research. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the sponsors such as the NSF.

APPENDIX: BLOCK DIAGRAMS OF TATE PAIRING HARDWARE UNIT

Figures 6 and 7 present the complete hardware schematics for the proposed Tate bilinear map pairing implementation.

REFERENCES

[1] [n. d.]. MIRACL Cryptographic SDK: Multiprecision Integer and Rational Arithmetic Cryptographic Library. https://github.com/miracl/MIRACL. Accessed: 2023-12-30.

[2] [n. d.]. Verilator: Verilog and SystemVerilog simulator. https://www.veripool.org/verilator/. Accessed: 2023-12-30.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[4] Dan Boneh and Brent Waters. 2007. Conjunctive, subset, and range queries on encrypted data. In *Proceedings of the 4th Conference on Theory of Cryptography* (Amsterdam, The Netherlands) *(TCC'07)*. Springer-Verlag, Berlin, Heidelberg, 535–554.

[5] D. R. Hankerson, S. A. Vanstone, and A. J. Menezes. 2011. *Guide to Elliptic Curve Cryptography*. Springer, New York.

[6] A. Karatsuba and Y. Ofman. 1963. Multiplication on many-digital numbers by automatic computers. *Translation in Physics-Doklady* 145, 2 (1963), 293–294.

[7] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shi-Feng Sun, Dongxi Liu, and Cong Zuo. 2018. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 745–762. https://doi.org/10.1145/3243734.3243753

[8] Hao Li, Jian Huang, Philip Sweany, and Dijiang Huang. 2008. FPGA implementations of elliptic curve cryptography and Tate pairing over a binary field. *Journal of Systems Architecture* 54, 12 (2008), 1077–1088. https://doi.org/10.1016/j.sysarc.2008.04.012

[9] Kealan McCusker, Noel O'Connor, and Dermot Diamond. 2006. Low-Energy Finite Field Arithmetic Primitives for Implementing Security in Wireless Sensor Networks. In *2006 International Conference on Communications, Circuits and Systems*, Vol. 3. 1537–1541. https://doi.org/10.1109/ICCCAS.2006.284964

[10] Dawn Xiaoding Song, D. Wagner, and A. Perrig. 2000. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. SP 2000.* 44–55. https://doi.org/10.1109/SECPRI.2000.848445

[11] Joachim von zur Gathen and Daniel Panario. 2001. Factoring Polynomials Over Finite Fields. *J. Symb. Comput.* 31, 1 (jan 2001), 3–17.

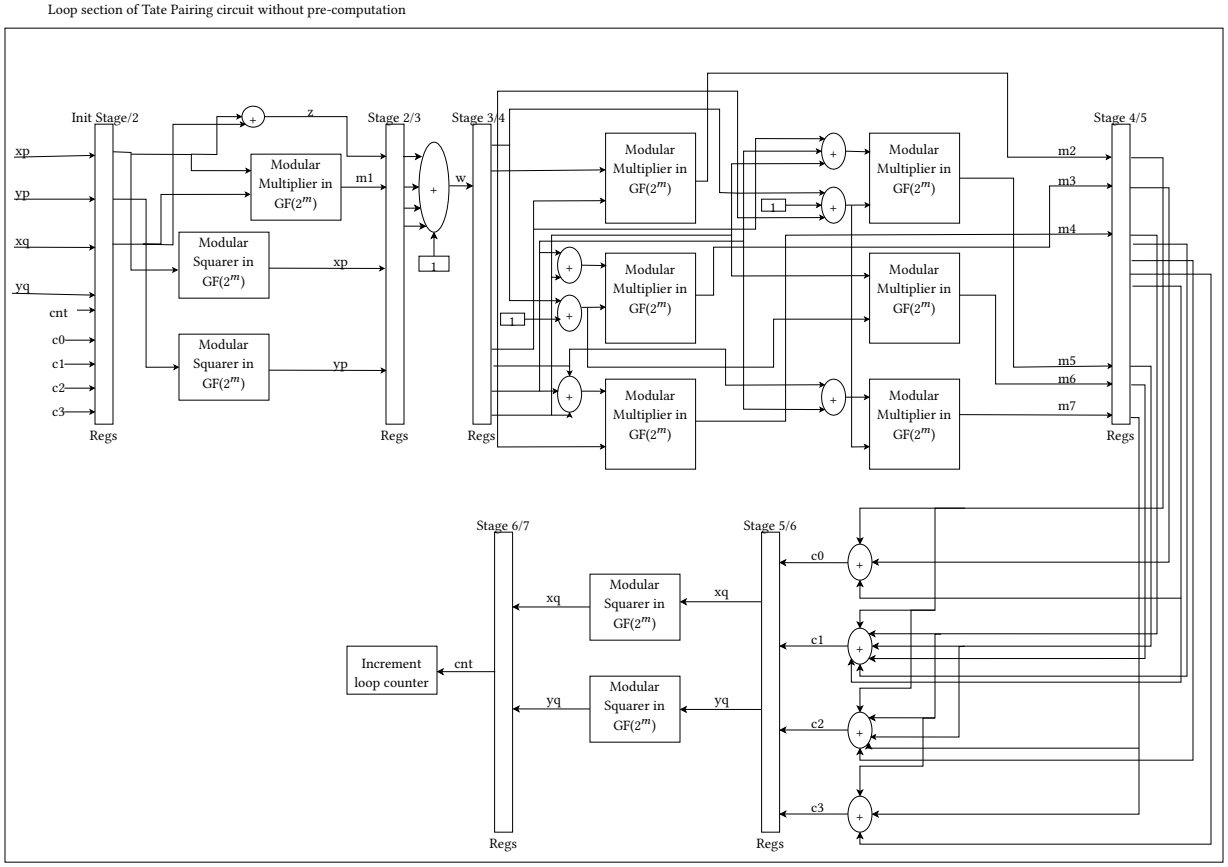

Figure 6: Schematic for the first section of Tate Pairing circuit

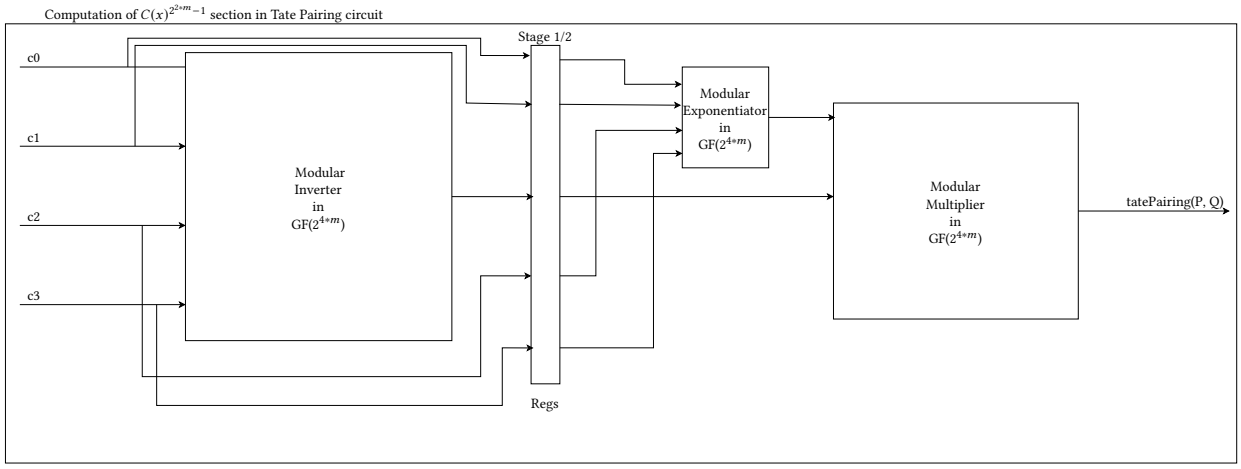

Figure 7: Schematic for the second section of Tate Pairing circuit