Instruction Set Extensions for Pairing-Based Cryptography*

Tobias Vejda[1], Dan Page[2], and Johann Großschädl[2]

[1] Institute for Applied Information Processing and Communications,
Graz University of Technology, Inffeldgasse 16a, A–8010 Graz, Austria
Tobias.Vejda@iaik.tugraz.at
[2] University of Bristol, Department of Computer Science,
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, U.K.
{page,johann}@cs.bris.ac.uk

Abstract. A series of recent algorithmic advances has delivered highly effective methods for pairing evaluation and parameter generation. However, the resulting multitude of options means many different variations of base field must ideally be supported on the target platform. Typical hardware accelerators in the form of co-processors possess neither the flexibility nor the scalability to support fields of different characteristic and order. On the other hand, extending the instruction set of a general-purpose processor by custom instructions for field arithmetic allows to combine the performance of hardware with the flexibility of software. To this end, we investigate the integration of a tri-field multiply-accumulate (MAC) unit into a SPARC V8 processor core to support arithmetic in $\mathbb{F}_p$, $\mathbb{F}_{2^n}$ and $\mathbb{F}_{3^n}$. Besides integer multiplication, the MAC unit can also execute dedicated multiply and MAC instructions for binary and ternary polynomials. Our results show that the tri-field MAC unit adds only a small size overhead while significantly accelerating arithmetic in $\mathbb{F}_{2^n}$ and $\mathbb{F}_{3^n}$, which sheds new light on the relative performance of $\mathbb{F}_p$, $\mathbb{F}_{2^n}$ and $\mathbb{F}_{3^n}$ in the context of pairing-based cryptography.

1 Introduction

Although pairings, or bilinear maps, on elliptic curves were initially only useful as a destructive tool for cryptanalysis, a slew of constructive applications [12] has motivated research into efficient pairing evaluation. Clearly the dominant form of optimisation for pairing evaluation lies at the algorithmic level; for a good overview of the evolution of optimisations, see the description of Scott [37]. In short, improvement of seminal but unpublished work by Miller [31] resulted in the first practical algorithms for evaluation of the Tate pairing [7,16]. These

* The work described in this paper has been supported by the European Commission through the IST Programme under contract no. IST-2002-507932 ECRYPT. The information in this paper reflects only the authors' views, is provided as is, and no guarantee or warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

T. Takagi et al. (Eds.): Pairing 2007, LNCS 4575, pp. 208–224, 2007.

results were further optimised by Duursma and Lee [13] who developed an inexpensive, closed form for specific parameterisations, later improved by Kwon [30]. Their techniques were generalised and extended to produce the Eta [6] and Ate [26] pairings, currently considered the fastest means of evaluating cryptographic pairings.

Much like the situation with vanilla elliptic curve cryptography (ECC), there is a broad range of choices to consider when actually implementing these algorithms; this range is amplified by the larger number of parameterisation options [37]. There are three major regions within the hardware/software design space for pairing evaluation, each offering a different cost versus performance trade-off. At one end of the design space, the composition of arithmetic in $\mathbb{F}_{q^k}$ has motivated numerous designs for hardware accelerators [42,10,27,28,8], which follow high-performance ECC accelerator design by utilising dedicated, parallel execution units. Coupled with the potential for pipelining, this approach trades area in favour of low latency. One can make the opposite trade-off by following ECC co-processor design and utilising a single execution unit in a more iterative manner [35,17]. With the co-processor coupled to a general-purpose processor core, this approach moves toward a more flexible solution in that it can, for example, be used to accelerate curve and field operations that are both vital within pairing-based protocols. Both these approaches have lent to some extent on efficient hardware implementation [34,9,19] of arithmetic in $\mathbb{F}_q$. Finally, and at the other end of the design spectrum, one can consider implementation entirely in software. Utilising niche techniques for representation and arithmetic in $\mathbb{F}_{3^m}$ [25,3] and better known techniques drawn from experience with ECC for $\mathbb{F}_{2^n}$ and $\mathbb{F}_p$, one can efficiently evaluate pairings on general-purpose desktop [37] and embedded [38] processors.

Somewhere between purely software and co-processor-assisted implementation lies the technique of *instruction set extension* (ISE) [22]. The premise here is that after careful workload characterisation, it is possible to identify a small set of operations that dominate performance in a software implementation. By supporting these specific operations using additional or modified hardware and exposing their behaviour to the programmer via the instruction set architecture (ISA), performance can be significantly improved. This is possible with only minor penalties in terms of datapath disruption and logic overhead. In the context of ECC, the use of ISEs has focused on acceleration of arithmetic in $\mathbb{F}_q$; see for example [29,21]. Since the efficacy of $\mathbb{F}_q$ underpins the performance of pairing evaluation, one can clearly reuse this work to gain an advantage. Moreover, as pairings can be parameterised by several different types of base field, one can leverage the advantages of unified multiplier circuits [36,20,2].

To summarise, the implementation of a means for pairing evaluation depends on arithmetic in $\mathbb{F}_q$; if the performance of said arithmetic can be improved, one can expect incremental but non-trivial improvements in the cost of pairing evaluation. Our goal in this paper is the construction of an ISE for the SPARC V8 compliant LEON-2 processor that enables acceleration of pairings parameterised over $\mathbb{F}_{2^n}$, $\mathbb{F}_{3^n}$ and $\mathbb{F}_p$ for large p. To this end, we developed an efficient tri-field

arithmetic unit based on *redundant signed digit* (RSD) encoding [4]. Algorithms and hardware architectures for long integer modular multiplication using RSD encoding were first proposed by Takagi et al. [39,40]. Other work in this area includes that of Öztürk et al. [33], who designed a tri-field Montgomery multiplier; we are careful to compare and contrast our results with their work. The major advantage of hardware acceleration through instruction set extensions is the ability to use optimised algorithms for fast squaring (resp. fast cubing) in $\mathbb{F}_{2^n}$ (resp. $\mathbb{F}_{3^n}$). This is significant since squaring (resp. cubing) is fundamental to the overall performance of pairing evaluation.

The rest of this paper is organised as follows. In Section 2 we present an overview of cryptographic pairings. Our aim is to construct a flexible cost model to use in comparisons of performance; this is presented in Appendix A. Section 3 details algorithms for arithmetic in the three fields $\mathbb{F}_{2^n}$, $\mathbb{F}_{3^n}$ and $\mathbb{F}_p$ and Section 4 introduces our tri-field multiplier design. Section 5 describes the LEON-2 processor and the modifications required to accommodate the tri-field multiplier before a set of experimental results are analysed in Section 6. Finally, we present some conclusions in Section 7 that demonstrate a clear improvement in arithmetic and pairing performance for all three fields.

2 Cryptographic Pairings

Let E be an elliptic curve over a finite field $\mathbb{F}_q$, and let $\mathcal{O}$ denote the identity element of the associated group of rational points $E(\mathbb{F}_q)$. For a positive integer $l|\#E(\mathbb{F}_q)$ co-prime to q, let k be the minimal positive integer such that $l|(q^k-1)$; k is often called the embedding degree or security multiplier. Let $E(\mathbb{F}_q)[l]$ denote the subgroup of $E(\mathbb{F}_q)$ of all points whose order is divisible by l, and similarly for the degree k extension of $\mathbb{F}_q$. Thus, the Tate pairing of order l is a map from elements of two source groups to a target group

$$e_l : E(\mathbb{F}_q)[l] \times E(\mathbb{F}_{q^k})[l] \rightarrow \mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^l$$

which satisfies the following properties

- For each $P \neq \mathcal{O}$ there exists $Q \in E(\mathbb{F}_{q^k})[l]$ such that $e_l(P,Q) \neq 1 \in \mathbb{F}^*_{q^k}/(\mathbb{F}^*_{q^k})^l$.
- For any integer n, $e_l([n]P,Q) = e_l(P,[n]Q) = e_l(P,Q)^n$ for all $P \in E(\mathbb{F}_q)[l]$ and $Q \in E(\mathbb{F}_{q^k})[l]$.
- Let $L = hl$. Then $e_l(P,Q)^{(q^k-1)/l} = e_L(P,Q)^{(q^k-1)/L}$.
- It is efficiently computable.

Since e_l is defined as taking $P \in E(\mathbb{F}_q)[l]$ and $Q \in E(\mathbb{F}_{q^k})[l]$ as input, it is common to define a distortion map Ψ to lift elements of $E(\mathbb{F}_q)[l]$ into elements of $E(\mathbb{F}_{q^k})[l]$; careful selection of the map permits specific optimisation within the algorithm to evaluate the pairing. Selection of parameters involves many subtle trade-offs between security and performance in source and target groups; it is far from clear which is the single best parameterisation. Ignoring issues of curve

generating and concentrating on performance of the pairing (instead of curve operations), one might opt to compare different selections for q by balancing the target group size. For example

$$\begin{array}{lcll} q = 2^m & \rightarrow & k = 4, m = 233 & \rightarrow \quad \log_2(q^k) \sim 932 \\ q = 3^m & \rightarrow & k = 6, m = 97 & \rightarrow \quad \log_2(q^k) \sim 922 \\ q = p & \rightarrow & k = 6, \log_2(p) \sim 160 & \rightarrow \quad \log_2(q^k) \sim 960. \end{array}$$

In reality, these parameters probably provide slightly less than the level of security typically required. However, they allow us to make somewhat reasoned comparisons later on; we derive a rough cost estimate for pairing evaluation in Appendix A.

3 Field Arithmetic

In this section we briefly review the basic algorithms for arithmetic operations in $\mathbb{F}_p$, $\mathbb{F}_{2^m}$, and $\mathbb{F}_{3^m}$. The elements of a prime field $\mathbb{F}_p$ can be represented by the integers from 0 to $p-1$. On the other hand, the elements of extension fields of characteristic two and three are commonly represented by binary and ternary polynomials, respectively. Addition and multiplication in $\mathbb{F}_p$ is performed modulo the prime p, while arithmetic in $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$ is carried out modulo an irreducible polynomial of degree exactly m with coefficients from the respective base field ($\mathbb{F}_2$ or $\mathbb{F}_3$).

Due to the large field orders used in pairing-based cryptography, the field elements can not be directly processed on a processor with a 32 or 64-bit datapath. Software implementations usually solve the mismatch between the operand length and the size of the processor datapath by storing the field elements in arrays of single-precision words (e.g. arrays of 32-bit unsigned integers) and using efficient arithmetic algorithms that manipulate these arrays with help of the instructions provided by the processor, e.g. (32×32)-bit multiply instructions [11]. In the following, we will denote the wordsize of the processor by w and the number of w-bit words required to store an n-bit field element by t, i.e. we have $t = \lceil n/w \rceil$. For example, an n-bit integer A can be stored in an array of t words, each consisting of w bits: $A = (A_{t-1}, \ldots, A_1, A_0)$. The words A_{t-1} and A_0 are the most and least significant word of A, respectively. Equation (1) specifies the relation between the integer A and the w-bit words A_i.

$$A = \sum_{i=0}^{t-1} A_i \cdot 2^{i \cdot w} = A_{t-1} \cdot 2^{(t-1) \cdot w} + \cdots + A_1 \cdot 2^w + A_0 \tag{1}$$

The elements of $\mathbb{F}_{2^m}$ are polynomials of degree up to $m-1$ with coefficients from $\mathbb{F}_2 = \{0, 1\}$. A binary polynomial of degree $m-1$ can be represented by a bitstring of length m in which each bit corresponds to a coefficient. This bitstring can be stored in an array of w-bit words, similar to a long integer. The same holds for elements of $\mathbb{F}_{3^m}$, but it must be considered that two bits are necessary for each coefficient of a ternary polynomial. Therefore, the number of w-bit words required for storing a ternary polynomial of degree $m-1$ is $t = \lceil 2m/w \rceil$.

3.1 Addition and Subtraction

Both addition and subtraction of two elements of a prime field is straightforward to implement. The long integers representing the field elements are added/subtracted, followed by a reduction modulo the prime p if the result is not within the interval $[0, p-1]$. This reduction is simply done by subtracting or adding p. Many processor architectures, including SPARC, feature an add-with-carry and subtract-with-borrow instruction to facilitate long integer arithmetic.

Addition of elements of $\mathbb{F}_{2^m}$ is simply a logical XOR operation and does not require a reduction. Addition and subtraction in ternary extension fields is more complex than addition in the two other field types [25]. An addition of ternary polynomials requires to add the coefficients modulo 3, which is relatively costly since this operation is not supported by general-purpose processors.

3.2 Multiplication and Squaring/Cubing

Multiplication in $\mathbb{F}_p$ is done by multiplying the two integers representing the field elements and then reducing the product modulo the prime p. Long integer multiplication can be performed through operand scanning or product scanning [24]. In the following we will discuss one method in more detail, namely product scanning, which is based on Comba's multiplication technique [11].

Algorithm 1. Comba multiplication [11].

Input: Two t-word integers $A = (A_{t-1}, \ldots, A_1, A_0)$ and $B = (B_{t-1}, \ldots, B_1, B_0)$.
Output: The $2t$-word product $C = A \cdot B = (C_{2t-1}, \ldots, C_1, C_0)$.

1: $S \leftarrow 0$
2: **for** i from 0 to $t-1$ **do**
3: **for** j from 0 to i **do**
4: $S \leftarrow S + A_j \cdot B_{i-j}$
5: **end for**
6: $C_i \leftarrow S \bmod 2^w$
7: $S \leftarrow S/2^w$
8: **end for**
9: **for** i from t to $2t-2$ **do**
10: **for** j from $i-t+1$ to $t-1$ **do**
11: $S \leftarrow S + A_j \cdot B_{i-j}$
12: **end for**
13: $C_i \leftarrow S \bmod 2^w$
14: $S \leftarrow S/2^w$
15: **end for**
16: $C_{2t-1} \leftarrow S \bmod 2^w$

Comba's method [11] for multiple-precision multiplication is shown in Algorithm 1. It has a nested loop structure and accumulates the inner-product terms $A_j \cdot B_{i-j}$ on a column-by-column basis. The operation carried out in the

inner loop of this algorithm is a *multiply-and-accumulate* operation. That is, two w-bit words A_i, B_j are multiplied and the $2w$-bit product is then added to a running sum S. As this sum can become $2w + \lceil \log_2(t) \rceil$ bits long, three word-size registers are needed to hold it during the computation [24]. Algorithm 1 performs a total of t^2 single-precision multiplications when A and B consist of t words.

Long integer squaring is a special case of long integer multiplication and allows for optimisation to reduce the execution time. Algorithm 1 can be rewritten such that only $(t^2 + t)/2$ single-precision multiplications to need to be carried out when $A = B$, which corresponds to a reduction of almost 50% compared to the t^2 single-precision multiplications when multiplying two distinct integers. In practice, however, squaring is only slightly faster than multiplication.

Multiplication in $\mathbb{F}_{2^m}$ requires to multiply two binary polynomials, followed by a reduction modulo an irreducible polynomial. A well-known algorithm for polynomial multiplication over $\mathbb{F}_2$ is the so-called *shift-and-xor* method [24]. The multiplicand is scanned coefficient-wise and the multiplier added to a running sum, depending on the value of the coefficient. More advanced methods, like the *right-to-left comb* method [24], use look-up tables to reduce the number of shifts and xor operations. However, if the target processor provides an instruction for word-level multiplication of binary polynomials, then two elements of $\mathbb{F}_{2^m}$ can be multiplied in a similar way as shown in Algorithm 1. Squaring of a binary polynomial is a linear operation and hence much faster than multiplication.

Multiplication in $\mathbb{F}_{3^m}$ can be performed in a similar way as in $\mathbb{F}_{2^m}$, namely through polynomial multiplication and reduction modulo an irreducible polynomial. All algorithms for the multiplication of binary polynomials, ranging from the shift-and-xor method to the left-to-right comb method, can be adapted to work for the multiplication of ternary polynomials as well. Moreover, Comba's multiplication technique could also be used for multiplying ternary polynomials if the processor provides a suitable word-level multiply instruction. Unfortunately, this is not the case for today's general-purpose processors, but a custom instruction for word-level multiplication of ternary polynomials can be easily integrated into any standard RISC architecture, as will be demonstrated in the following sections. Cubing a ternary polynomial is a linear operation, similar to squaring of a binary polynomial, and thus much faster than multiplication.

3.3 Modular Reduction

A widely-used algorithm for modular reduction of integers is due to Montgomery [32]. Montgomery's algorithm is a generic modular reduction method that works for any odd prime. However, certain primes, like pseudo-Mersenne primes or generalised-Mersenne primes, facilitate faster reduction methods. A reduction modulo such special primes can be performed efficiently with additions and shift operations [24].

Polynomial modular reduction is also very fast if the irreducible polynomial has few non-zero coefficients. Of particular importance are irreducible trinomials and pentanomials since they allow to accomplish a reduction with simple shift operations and polynomial additions (see [24] for further details).

4 Tri-field Multiplier

Recent research [36,20,2] has provided a number of so-called unified multipliers that reuse elements of the datapath for different field types. They rely on multi-field adder cells (e.g. dual-field adders) as basic building block. Most previous work focussed on the unification of integers and binary polynomials. The first unified multiplier for integers, binary and ternary polynomials was presented by Öztürk et al. [33]. Multiplier designs making use of unified components are efficient in terms of silicon area as the adder cells are reused for different types of operands. We exploit the advantages of this approach and introduce a unified multiply-accumulate (MAC) unit able to support our custom instruction set.

4.1 Redundant Signed Digit (RSD) Representation

The design of fast parallel multipliers relies on efficient arithmetic that restricts the carry propagation in addition to a few positions. Previous work examined the use of (3:2) counters or (4:2) compressors and *carry-save* representation. A different approach is to use a signed representation. Instead of splitting the sum of two numbers into carry and sum vectors, each number is split into a positive and a negative part. This representation is called *borrow-save* [5] and follows the relation

$$\mathbf{X} = \sum_{i=0}^{n-1} x_i \cdot 2^i = \sum_{i=0}^{n-1} (x_i^+ - x_i^-) \cdot 2^i \tag{2}$$

where $\mathbf{X}$ denotes an n-digit number with digit set $\{-1, 0, 1\}$. Such digit systems were first studied by Avizienis [4] and are called *redundant signed digit* (RSD) systems. This digit set gives an advantage over the two's complement form as it naturally handles signed numbers, which simplifies the design of a multiplier for signed and unsigned integers. Furthermore, the RSD system allows a straightforward representation of the coefficients of ternary polynomials.

RSD addition of radix-2 numbers is performed in two steps whereby in each step a carry can occur. Hence, the maximum carry propagation distance is restricted to two digits, which facilitates efficient hardware implementation. Our design of the RSD adder performs the addition of integers in the conventional way following the rules of RSD arithmetic. The addition of binary polynomials uses a different recoding to prevent carry propagation. Ternary polynomials, on the other hand, are recoding after each step to ensure that the coefficients of the result remain in $\mathbb{F}_3$.

4.2 Architecture of the Multiply-Accumulate Unit

Figure 1 shows the main components of our unified MAC unit. It consists of a unified multiplier followed by a unified adder acting as accumulator. The MAC unit is pipelined, and hence we need to store the result of the multiplier for one cycle. Our design is scalable to any operand length and allows for different trade-offs between silicon area and critical path delay. We explored two specific

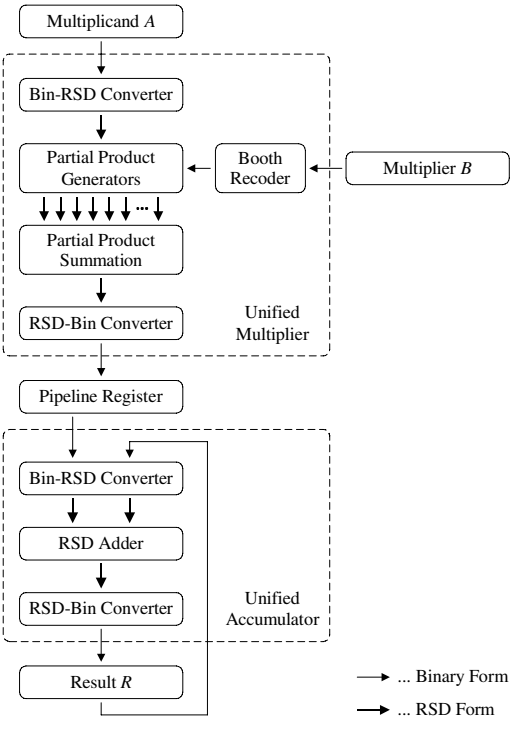

Fig. 1. Main components of the unified multiply-accumulate unit

implementations of the MAC unit; the first contains a (32×16)-bit multiplier and the second a smaller (32×8)-bit multiplier. These multipliers perform a full (32×32)-bit multiplication in two and four clock cycles, respectively.

Parallel multipliers can be implemented in from of an array structure or a tree structure. Array multipliers feature high regularity and short interconnect wires, while tree multipliers have a smaller critical path delay. When using RSD representation, this trade-off can be crucial since a single RSD adder cell is more complex than a conventional full adder. Due to negatively weighted inputs to adders, common techniques developed for the summation of partial products are difficult to apply in this setting. We implemented both the array and the tree architecture to assess the resulting area and delay complexities.

As shown in Figure 1, the result of the multiplication is converted from RSD to binary representation before it is buffered in the pipeline register, and then converted back into RSD form. These conversions increase the critical path, but reduce the silicon area since the binary representation requires only half of the storage that would be needed for RSD representation. A typical standard-cell implementation of the MAC unit has a delay between 12 and 22 ns, depending on the architecture and size of the multiplier (see Section 6 for details). These delays correspond to a maximum clock frequency between 45 and 83 MHz, which

is reasonable for embedded devices like smart cards. Higher clock frequencies are possible through the integration of additional pipeline registers.

5 Extended LEON-2 Processor

SPARC V8 [43] is a general-purpose RISC architecture with a 32-bit datapath and a "windowed" register file containing an implementation-dependent number of general-purpose registers (GPRs), of which 32 are visible to the programmer at a time. Besides the GPRs, the SPARC V8 architecture also includes several special-purpose registers like the multiply-divide register (`%y`) and a total of 31 ancillary state registers (`%asr1` to `%asr31`).

The SPARC instruction set contains Delayed Control Transfer Instructions (DCTI). In particular, branches and calls have an architectural delay slot of one instruction, which means that the instruction immediately following a DCTI is executed (unless the DCTI annuls it) before the control transfer to the target address is completed. Arithmetic and logical instructions have a conventional three-operand format with two source registers and a single destination register [43]. Multiply instructions like `smul` and `umul` write the 32 least significant bits of the product to a destination register and the 32 most significant bits to the multiply-divide register (`%y`). The `rdy` instruction allows to transfer the content of register `%y` to a GPR.

5.1 Main Characteristics of the LEON-2 Core

The LEON-2 processor [15] is a configurable and synthesizable VHDL implementation of the SPARC V8 instruction set. Originally developed by the European Space Agency, the LEON-2 softcore is now maintained by Gaisler Research and has found widespread use in system-on-chip (SOC) designs in recent years. The LEON-2 VHDL model is highly configurable; various options like the number of register windows, the size and organisation of caches, and performance/area trade-offs for the integer multiplier can be defined through a single configuration file. In addition, the LEON-2 core is extensible since the full VHDL source code is available under the GNU LGPL license.

The LEON-2 pipeline can be configured to have either one or two load delay cycles. We used a LEON-2 core with one load delay cycle as this configuration allows to achieve better performance in FPGAs. The LEON-2 processor also contains a hardware multiplier that can be configured to perform a (32×32)-bit integer multiplication in either 35, 4, 2, or 1 clock cycles.

5.2 Custom Instructions

The extensions for pairing-based cryptography we propose in this paper include a total of five custom instructions to accelerate arithmetic operations in prime fields, binary fields, and ternary fields. Table 1 gives an overview of the instructions and summarises the operations they perform. The first two instructions

Table 1. Custom instructions for pairing-based cryptography

Format	Description	Operation
`umac rs1, rs2`	Unsigned Mul. and Acc.	$accu \leftarrow accu + rs1 \times rs2$
`umac2 rs1, rs2`	Unsigned Mul. and Acc. Twice	$accu \leftarrow accu + 2(rs1 \times rs2)$
`shacr rd`	Shift Accu Registers Right	$rd \leftarrow accu[31:0]\,;\ accu \leftarrow accu \gg 32$
`gf2mac rs1, rs2`	Binary Poly. Mul. and Acc.	$accu \leftarrow accu \oplus rs1 \otimes rs2$
`gf3mac rs1, rs2`	Ternary Poly. Mul. and Acc.	$accu \leftarrow accu + rs1 \times rs2$

(`umac` and `umac2`) allow to speed up the inner loop operations of long integer multiplication and squaring according to Comba's algorithm. The instructions `gf2mac` and `gf3mac` can be used to implement the multiplication of binary and ternary polynomials, respectively.

The `umac` instruction performs a MAC operation on unsigned 32-bit integers. More precisely, `umac` multiplies the content of two GPRs, treating both operands as unsigned integers, and adds the 64-bit product to a cumulative sum stored in the three registers `%asr20`, `%y`, and `%asr18`, subsequently called *accu registers*. The cumulative sum is, in general, exceeding 64 bits in precision when several 64-bit products are summed up. Therefore, three 32-bit registers are needed to accommodate the cumulative sum, whereby the 32 least significant bits are stored in `%asr18`, the bits 32 through 63 in register `%y`, and the most significant bits in `%asr20`, respectively. After adding the 64-bit product to the cumulative sum, the result is written back to the accu registers (see Figure 2). The custom instruction `shacr` allows to shift the cumulative sum held in the accu registers 32 bits to the right, whereby the least significant 32-bit word of the cumulative sum (i.e. the content of `%asr18`) is written to a destination register `rd`.

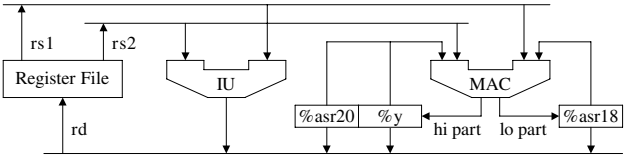

Fig. 2. Datapath consisting of integer unit (IU) and MAC unit with accu registers

We implemented a "pairing-friendly" MAC unit for the LEON-2 core consisting of a (32×16)-bit tree multiplier and a 72-bit accumulator. The 72-bit accumulator guarantees that up to 256 double-precision (i.e. 64-bit) products can be summed up without overflow or loss of precision, which is sufficient for cryptographic applications. Besides the custom instructions shown in Table 1, the MAC unit is also capable to execute the "native" SPARC multiply instructions like `umul` and `smul` [23]. Therefore, the proposed extensions for pairing-based cryptography can be easily integrated into the LEON-2 core by simply replacing the integer multiplier with a MAC unit that provides the extra functionality. In

addition to modifications of the LEON-2 core, we also adapted the tool-chain, in particular the GNU assembler `gas`, to support the custom instructions.

A LEON-2 core equipped with a (32×16+72)-bit MAC unit executes the "native" SPARC V8 multiply instructions `smul/umul` in two clock cycles, whereby higher part of the product is written to the `%y` register, while the lower part is directed to a GPR in the register file. The custom instruction `umac` also has a latency of two cycles, but places its result in the accu registers (and not in a GPR), and therefore an independent instruction can be executed in the integer unit during the second cycle of a `umac` instruction [23]. This parallel execution is possible since the buses connecting the register file and the functional units are not occupied during the second cycle of a `umac` instruction, similar to the execution of the `madd` instruction in MIPS32 processors.

6 Experimental Results

We prototyped the extended LEON-2 processor on a Xess XSV800 board which houses a Xilinx Virtex FPGA providing about 800k gates. The LEON-2 source code contains scripts and configuration files for several FPGA boards, including the XSV800. We used Xilinx XST 8.3 to perform the synthesis. As mentioned in Section 4, we implemented several versions of the MAC unit with different multiplier dimensions (32×16 bit, 32×8 bit) and structures (array, tree). In order to assess area and delay of the different implementations, we synthesised the MAC unit not only as part of the LEON-2, but also as stand-alone circuit using a 0.35 μm standard cell library. The results of these synthesis runs are summarised in Table 2. The gate equivalents were calculated taking a 2-NAND gate from the same library as reference. This gate has an area of 55 μm^2.

Table 2. Synthesis results of different implementations of the MAC unit

MAC Type	Silicon Area	Delay	Max. Frequency	Latency
(32×16)-array	16,400 GE	22 ns	45 MHz	2 cycles
(32×16)-tree	16,200 GE	16 ns	62 MHz	2 cycles
(32×8)-array	11,900 GE	14 ns	71 MHz	4 cycles
(32×8)-tree	12,700 GE	12 ns	83 MHz	4 cycles

Besides area and delay, we also evaluated the performance gain due to the integration of the MAC unit and the custom instructions. Table 3 and 4 summarise the execution times of arithmetic operations in $\mathbb{F}_p$, $\mathbb{F}_{2^m}$, and $\mathbb{F}_{3^m}$ when using native SPARC instructions and the extended instruction set, respectively. Our reference implementation (Table 3) is based on a ANSI C library for arithmetic in $\mathbb{F}_p$, $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$ that was developed at the University of Bristol. It uses Montgomery multiplication in $\mathbb{F}_p$ and features a number of Assembler macros for performance-critical operations. The multiplication of binary polynomials is performed via a recursive Karatsuba technique, while the polynomial squaring

Table 3. Arithmetic performance (in clock cycles) of the SPARC V8 instruction set

Field Arithmetic in $\mathbb{F}_{2^n}$			
Field	Addition	Multiplication	Squaring
$\mathbb{F}_{2^{163}}$	162	4,978	1,468
$\mathbb{F}_{2^{233}}$	197	7,010	1,463
$\mathbb{F}_{2^{283}}$	215	14,531	1,861
$\mathbb{F}_{2^{353}}$	275	14,154	2,359
$\mathbb{F}_{2^{457}}$	323	21,496	2,840
$\mathbb{F}_{2^{557}}$	413	42,863	3,264
Field Arithmetic in $\mathbb{F}_{3^m}$			
Field	Addition	Multiplication	Cubing
$\mathbb{F}_{3^{79}}$	96	8,992	1,360
$\mathbb{F}_{3^{97}}$	116	13,460	1,448
$\mathbb{F}_{3^{163}}$	156	26,779	1,996
$\mathbb{F}_{3^{193}}$	176	35,690	2,251
$\mathbb{F}_{3^{239}}$	196	39,855	2,448
$\mathbb{F}_{3^{353}}$	276	79,195	3,650
Field Arithmetic in $\mathbb{F}_p$			
Field	Addition	Multiplication	Squaring
$\mathbb{F}_p, \log_2(p) = 160$	69	1,183	1,183
$\mathbb{F}_p, \log_2(p) = 192$	80	1,334	1,334
$\mathbb{F}_p, \log_2(p) = 224$	92	1,618	1,618
$\mathbb{F}_p, \log_2(p) = 256$	104	1,907	1,907
$\mathbb{F}_p, \log_2(p) = 384$	152	3,866	3,866
$\mathbb{F}_p, \log_2(p) = 512$	200	6,254	6,089

is done via table look-up. Ternary polynomials use a bit-sliced representation where high and low parts of the coefficients are stored in separate vectors. Multiplication of ternary polynomials is also based on the Karatsuba approach and cubing on the table look-up method. The addition of ternary polynomials is realised in a straightforward way. We compiled the library using a GCC cross compiler for the SPARC V8 architecture with optimisations enabled. All timings shown in Table 3 were measured under warm cache conditions with help of the built-in cycle counter (register `%cycnt`) of the modified LEON-2 core.

Table 4 shows the cycle counts of the arithmetic operations when using our custom instructions described in Section 5. In short, the custom instructions accelerate multiplication in $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$ by a factor of (at least) 10 and 20, respectively, while multiplication in $\mathbb{F}_p$ achieves a two-fold performance gain.

7 Conclusions

Considering the results from the previous section in the context of approximate costs for pairing evaluation given in Appendix A, the value of ISE and our tri-field MAC unit is clear. Specifically, for similar sized base fields outlined in

Table 4. Arithmetic performance (in clock cycles) of the extended instruction set

Field Arithmetic in $\mathbb{F}_{2^n}$			
Field	Addition	Multiplication	Squaring
$\mathbb{F}_{2^{163}}$	65	415	166
$\mathbb{F}_{2^{233}}$	81	634	207
$\mathbb{F}_{2^{283}}$	89	778	244
$\mathbb{F}_{2^{353}}$	113	1,238	311
$\mathbb{F}_{2^{457}}$	137	1,806	378
$\mathbb{F}_{2^{557}}$	161	2,545	508
Field Arithmetic in $\mathbb{F}_{3^m}$			
Field	Addition	Multiplication	Cubing
$\mathbb{F}_{3^{79}}$	65	422	475
$\mathbb{F}_{3^{97}}$	85	637	593
$\mathbb{F}_{3^{163}}$	125	1,257	927
$\mathbb{F}_{3^{193}}$	125	1,295	984
$\mathbb{F}_{3^{239}}$	165	2,069	1,261
$\mathbb{F}_{3^{353}}$	245	4,247	1,896
Field Arithmetic in $\mathbb{F}_p$			
Field	Addition	Multiplication	Squaring
$\mathbb{F}_p, \log_2(p) = 160$	68	498	498
$\mathbb{F}_p, \log_2(p) = 192$	80	574	574
$\mathbb{F}_p, \log_2(p) = 224$	92	759	759
$\mathbb{F}_p, \log_2(p) = 256$	104	913	913
$\mathbb{F}_p, \log_2(p) = 384$	152	1,883	1,883
$\mathbb{F}_p, \log_2(p) = 512$	200	3,094	3,094

Section 2 (and ignoring issues of curve generation) we find that the number of clock cycles required to evaluate a pairing in $\mathbb{F}_{2^n}$ is reduced from $6,762,354$ to $646,554$, in $\mathbb{F}_{3^n}$ from $12,061,117$ to $981,984$, and in $\mathbb{F}_p$ from $10,788,960$ to $4,541,760$. Clearly these are only estimates, but they equate to between a two and twelve-fold saving depending on the field choice, a significant improvement for such little overhead. Further, utilising the tri-field MAC greatly reduces the cost ratio between $\mathbb{F}_{3^n}$ and other choices. Previously this parameterisation was at least twice as slow as the selection of $\mathbb{F}_{2^n}$, for example, with this gap widening for larger n. However, with custom instructions and the integration of our tri-field MAC unit, the two choices are roughly comparable in performance.

The advantages of this result are two-fold. Firstly, with only minor modification to the processor datapath and ISA one can significantly improve the performance of pairing evaluation. Unlike some dedicated hardware accelerators, this improvement costs only a moderate size overhead and is useful in accelerating operations within the pairing, at the protocol level and within vanilla ECC. Secondly, the acceleration of pairings over $\mathbb{F}_{3^n}$, which were previously prohibitively slow, allows a more free choice of parameterisation: if this is a good choice for the application, it need not be restrictive in terms of performance.

References

1. Ahmadi, O., Hankerson, D., Menezes, A.: Formulas for Cube Roots in $\mathbb{F}_{3^m}$. Available at: `http://www.cacr.math.uwaterloo.ca/ajmeneze/publications/cuberoots.pdf`
2. Au, L.-S., Burgess, N.: Unified Radix-4 Multiplier for $GF(p)$ and $GF(2^n)$. In: Application-Specific Systems, Architectures and Processors (ASAP), pp. 226–236. IEEE Press, Los Alamitos (2003)
3. Austrin, P.: Efficient Arithmetic in Finite Fields of Small, Odd Characteristic. MSc Thesis, Royal Institute of Technology, Stockholm (2004)
4. Avizienis, A.: Signed-Digit Number Representations for Fast Parallel Arithmetic. IRE Transactions on Electronic Computers 10(9), 389–400 (1961)
5. Bajard, J.-C., Duprat, J., Kla, S., Muller, J.-M.: Some Operators for On-Line Radix-2 Computations. Journal of Parallel and Distributed Computing 22(2), 336–345 (1994)
6. Barreto, P.S.L.M., Galbraith, S., ÓhÉigeartaigh, C., Scott, M.: Efficient Pairing Computation on Supersingular Abelian Varieties. In: Cryptology ePrint Archive, Report 2004/375 (2004)
7. Barreto, P.S.L.M., Kim, H., Lynn, B., Scott, M.: Efficient Algorithms for Pairing-Based Cryptosystems. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 354–368. Springer, Heidelberg (2002)
8. Bertoni, G., Breveglieri, L., Fragneto, P., Pelosi, G.: Parallel Hardware Architectures for the Cryptographic Tate Pairing. In: Information Technology: New Generations (ITNG), pp. 186–191. IEEE Press, Los Alamitos (2006)
9. Bertoni, G., Guajardo, J., Kumar, S., Orlando, G., Paar, C., Wollinger, T.: Efficient $GF(p^m)$ Arithmetic Architectures for Cryptographic Applications. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 158–175. Springer, Heidelberg (2003)
10. Beuchat, J.-L., Shirase, M., Takagi, T., Okamoto, E.: An Algorithm for the η_T Pairing Calculation in Characteristic Three and its Hardware Implementation. In: Cryptology ePrint Archive, Report 2006/327 (2006)
11. Comba, P.G.: Exponentiation cryptosystems on the IBM PC. IBM Systems Journal 29(4), 526–538 (1990)
12. Dutta, R., Barua, R., Sarkar, P.: Pairing-Based Cryptographic Protocols: A Survey. In: Cryptology ePrint Archive, Report 2004/064 (2004)
13. Duursma, I., Lee, H.: Tate Pairing Implementation for Hyperelliptic Curves $y^2 = x^p - x + d$. In: Laih, C.-S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 111–123. Springer, Heidelberg (2003)
14. Fong, K., Hankerson, D., López, J., Menezes, A.: Field Inversion and Point Halving Revisited. Technical Report CORR 2003-18, University of Waterloo (2003)
15. Gaisler, J.: The LEON-2 Processor User's Manual (Version 1.0.30) (July 2005), Available for download at `http://www.gaisler.com`
16. Galbraith, S., Harrison, K., Soldera, D.: Implementing the Tate pairing. In: Fieker, C., Kohel, D.R. (eds.) Algorithmic Number Theory (ANTS-V). LNCS, vol. 2369, pp. 324–337. Springer, Heidelberg (2002)
17. Grabher, P., Page, D.: Hardware Acceleration of the Tate Pairing in Characteristic Three. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 398–411. Springer, Heidelberg (2005)
18. Granger, R., Page, D., Smart, N.P.: High Security Pairing-Based Cryptography Revisited. In: Hess, F., Pauli, S., Pohst, M. (eds.) Algorithmic Number Theory (ANTS-VII). LNCS, vol. 4076, pp. 480–494. Springer, Heidelberg (2006)

19. Granger, R., Page, D., Stam, M.: Hardware and Software Normal Basis Arithmetic for Pairing Based Cryptography in Characteristic Three. IEEE Transactions on Computers 54(7), 852–860 (2005)
20. Großschädl, J.: A Bit-Serial Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 202–218. Springer, Heidelberg (2001)
21. Großschädl, J., Kumar, S., Paar, C.: Architectural Support for Arithmetic in Optimal Extension Fields. In: Application-Specific Systems, Architectures and Processors (ASAP), pp. 111–124. IEEE Press, Los Alamitos (2004)
22. Großschädl, J., Savaş, E.: Instruction Set Extensions for Fast Arithmetic in Finite Fields $GF(p)$ and $GF(2^m)$. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 133–147. Springer, Heidelberg (2004)
23. Großschädl, J., Tillich, S., Szekely, A.: Cryptography Instruction Set Extensions to the SPARC V8 Architecture. Preprint (submitted for publication, 2007)
24. Hankerson, D.R., Menezes, A.J., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer, Heidelberg (2004)
25. Harrison, K., Page, D., Smart, N.P.: Software Implementation of Finite Fields of Characteristic Three, for use in Pairing Based Cryptosystems. LMS Journal of Computation and Mathematics 5(1), 181–193 (2002)
26. Hess, F., Smart, N.P., Vercauteren, F.: The Eta Pairing Revisited. Transactions on Information Theory 52, 4595–4602 (2006)
27. Kerins, T., Marnane, W.P., Popovici, E.M., Barreto, P.S.L.M.: Efficient Hardware for the Tate Pairing Calculation in Characteristic Three. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 412–426. Springer, Heidelberg (2005)
28. Kerins, T., Popovici, E., Marnane, W.P.: Algorithms and Architectures for Use in FPGA Implementations of Identity Based Encryption Schemes. In: Becker, J., Platzner, M., Vernalde, S. (eds.) FPL 2004. LNCS, vol. 3203, pp. 74–83. Springer, Heidelberg (2004)
29. Kumar, S., Paar, C.: Reconfigurable Instruction Set Extension for Enabling ECC on an 8-Bit Processor. In: Becker, J., Platzner, M., Vernalde, S. (eds.) FPL 2004. LNCS, vol. 3203, pp. 586–595. Springer, Heidelberg (2004)
30. Kwon, S.: Efficient Tate Pairing Computation for Elliptic Curves over Binary Fields. In: Boyd, C., González Nieto, J.M. (eds.) ACISP 2005. LNCS, vol. 3574, pp. 134–145. Springer, Heidelberg (2005)
31. Miller, V.: Short programs for functions on curves. Available at: `http://crypto.stanford.edu/miller/miller.pdf`
32. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
33. Öztürk, E., Savas, E., Sunar, B.: A Versatile Montgomery Multiplier Architecture with Characteristic Three Support. Available at: `http://ece.wpi.edu/ sunar/preprints/versatile.pdf`
34. Page, D., Smart, N.P.: Hardware Implementation of Finite Fields of Characteristic Three. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 529–539. Springer, Heidelberg (2003)
35. Ronan, R., ÓhÉigeartaigh, C., Murphy, C., Scott, M., Kerins, T., Marnane, W.P.: An Embedded Processor for a Pairing-Based Cryptosystem. In: Information Technology: New Generations (ITNG), pp. 192–197. IEEE Press, Los Alamitos (2006)
36. Savas, E., Tenca, A.F., Koç, Ç.K.: A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 277–295. Springer, Heidelberg (2000)

37. Scott, M.: Implementing Cryptographic Pairings, Available at: ftp://ftp.computing.dcu.ie/pub/resources/crypto/pairings.pdf
38. Scott, M., Costigan, N., Abdulwahab, W.: Implementing Cryptographic Pairings on Smartcards. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 134–147. Springer, Heidelberg (2006)
39. Takagi, N., Yajima, S.: Modular Multiplication Hardware Algorithms with a Redundant Representation and their Application to RSA Cryptosystem. IEEE Transactions on Computers 41(7), 887–891 (1992)
40. Takagi, N.: A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation. IEEE Transactions on Computers 41(8), 949–956 (1992)
41. Shirase, M., Takagi, T., Okamoto, E.: Some Efficient Algorithms for the Final Exponentiation of η_T Pairing. In: Cryptology ePrint Archive, Report 2006/431 (2006)
42. Shu, C., Kwon, S., Gaj, K.: FPGA Accelerated Tate Pairing Based Cryptosystems over Binary Fields. In: Cryptology ePrint Archive, Report 2006/179 (2006)
43. SPARC International, Inc. The SPARC Architecture Manual Version 8 (August 1993) Available for download at http://www.sparc.org/standards/V8.pdf

A The Cost of Pairing Evaluation

We are generally interested in algorithms for pairing evaluation from the perspective of operation count rather than rigorous mathematical definition; this stems from our focus on implementation of arithmetic in $\mathbb{F}_q$. As such, we lean on other work to provide the most current, most efficient cost model. Let $\mathcal{A}_f$, $\mathcal{S}_f$, $\mathcal{C}_f$, $\mathcal{M}_f$ and $\mathcal{I}_f$ denote the cost of computing addition, squaring, cubing, multiplication and inversion in some finite field f. We quote a given cost model in terms of dominant operations; this typically means neglecting the cost of $\mathcal{A}_f$ for example, and concentrating on $\mathcal{M}_f$ and $\mathcal{I}_f$. Thanks to efficient methods for computing square roots [14] and cube roots [1] in $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$, respectively, we estimate their cost to be equivalent to squaring/cubing in the same fields; this makes sense given the usual opportunity to pre-compute square roots (resp. cube roots) via repeated squaring (resp. cubing).

We do not allow for pre-computation based on fixed values of either input to the pairing. Further, we are not concerned with compatibility, specifically we assume that if one utilises the Eta or Ate pairings their output need not be further powered to provide the same result as a comparable invocation of the Tate pairing.

Eta Pairing in Characteristic 2. Following Barreto et al. [6], consider the Eta pairing with source groups instantiated using the supersingular curve

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

over $\mathbb{F}_{2^m}$ where $b \in \{0, 1\}$ and the embedding degree $k = 4$. One can construct a tower of fields to form the required extension via $\mathbb{F}_{2^{2m}} = \mathbb{F}_{2^m}[\alpha]/(\alpha^2 - \alpha + 1)$ and $\mathbb{F}_{2^{4m}} = \mathbb{F}_{2^{2m}}[\beta]/(\beta^2 - \beta - \alpha)$. The distortion map $\Psi(x, y) = (\alpha^2 + x, \beta + \alpha x + y)$. The cost of evaluating the Eta pairing can be approximated by

$$(7(m + 1)/2)\mathcal{M}_{\mathbb{F}_{2^m}} + (4(m + 1)/2)\mathcal{S}_{\mathbb{F}_{2^m}}$$

while the final powering costs roughly

$$36\mathcal{M}_{\mathbb{F}_{2^m}} + \mathcal{I}_{\mathbb{F}_{2^{4m}}}.$$

Estimating $\mathcal{I}_{\mathbb{F}_{2^{4m}}} \sim 12\mathcal{M}_{\mathbb{F}_{3^m}}$, we get an overall cost of roughly

$$(3.5m + 51.5)\mathcal{M}_{\mathbb{F}_{2^m}} + (2m + 2)\mathcal{S}_{\mathbb{F}_{2^m}}.$$

For $m = 233$ this yields

$$867\mathcal{M}_{\mathbb{F}_{2^m}} + 468\mathcal{S}_{\mathbb{F}_{2^m}}.$$

Eta Pairing in Characteristic 3. Following Shirase et al. [41], consider the Eta pairing with source groups instantiated using the supersingular curve

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

over $\mathbb{F}_{3^m}$ where $b \in \{-1, +1\}$ and the embedding degree $k = 6$. One can construct a tower of fields to form the required extension via $\mathbb{F}_{3^{3m}} = \mathbb{F}_{3^m}[\rho]/(\rho^3 - \rho - b)$ and $\mathbb{F}_{3^{6m}} = \mathbb{F}_{3^{3m}}[\sigma]/(\sigma^2 + 1)$. The distortion map $\Psi(x, y) = (\rho - x, \sigma y)$. Shirase et al. [41][Table 2] show that evaluation of the Eta pairing with this parameterisation can cost as little as

$$(7.5m + 68.5)\mathcal{M}_{\mathbb{F}_{3^m}} + (8m + 8)\mathcal{C}_{\mathbb{F}_{3^m}} + \mathcal{I}_{\mathbb{F}_{3^{3m}}}$$

where they estimate $\mathcal{I}_{\mathbb{F}_{3^{3m}}} \sim 15.73\mathcal{M}_{\mathbb{F}_{3^m}}$ to get an overall cost of roughly

$$(7.5m + 84.23)\mathcal{M}_{\mathbb{F}_{3^m}} + (8m + 8)\mathcal{C}_{\mathbb{F}_{3^m}}.$$

For $m = 97$ this yields

$$811.73\mathcal{M}_{\mathbb{F}_{3^m}} + 784\mathcal{C}_{\mathbb{F}_{3^m}}.$$

Ate Pairing in Characteristic *p*. Using curve-specific optimisations, the Ate [26] pairing redefines the bilinear map as

$$\hat{e_l} : E(\mathbb{F}_p) \times \overline{E}(\mathbb{F}_{p^{k/2}}) \rightarrow \mathbb{F}^*_{p^k}$$

where $\overline{E}$ is the quadratic twist of an elliptic curve E defined over $\mathbb{F}_{p^{k/2}}$. As such, efficient arithmetic in $\mathbb{F}_{p^{k/2}}$ and $\mathbb{F}_{p^k}$ is required, both underpinned by arithmetic in $\mathbb{F}_p$. Granger et al. [18] describe a range of options for arithmetic in different instantiations of these fields; selection of $log_2(p) \sim 256$ and $k = 6$ implies Case A of [18][Section 5] which estimates the cost of pairing evaluation to be

$$9120\mathcal{M}_{\mathbb{F}_p}.$$