Designing an ASIP for Cryptographic Pairings over Barreto-Naehrig Curves

David Kammler*, Diandian Zhang*, Peter Schwabe†,
Hanno Scharwaechter*, Markus Langenberg‡, Dominik Auras*,
Gerd Ascheid*, Rainer Leupers*, Rudolf Mathar‡, Heinrich Meyr*

* *Institute for Integrated Signal Processing Systems (ISS),*
RWTH Aachen University, Aachen, Germany
Email: kammler@iss.rwth-aachen.de

† *Department of Mathematics and Computer Science,*
Eindhoven University of Technology, Eindhoven, Netherlands, previously at ‡
Email: peter@cryptojedi.org

‡ *Institute for Theoretical Information Technology (TI),*
RWTH Aachen University, Aachen, Germany
Email: mathar@ti.rwth-aachen.de

Abstract

This paper presents a design-space exploration of an application-specific instruction-set processor (ASIP) for the computation of various cryptographic pairings over Barreto-Naehrig curves (BN curves). Cryptographic pairings are based on elliptic curves over finite fields—in the case of BN curves a field $\mathbb{F}_p$ of large prime order p. Efficient arithmetic in these fields is crucial for fast computation of pairings. Moreover, computation of cryptographic pairings is much more complex than elliptic-curve cryptography (ECC) in general. Therefore, we facilitate programming of the proposed ASIP by providing a C compiler.

In order to speed up $\mathbb{F}_p$-arithmetic, a RISC core is extended with additional functional units. The critical path delay of these units is adjusted to the base architecture in order to maintain the operating frequency. Independently from that adjustment, these units are scalable allowing for a trade-off between execution time and area consumption. Because the resulting speedup can be limited by the memory throughput, utilization of multiple data memories is proposed. However, developing a C compiler for multiple memories is a challenging task. Therefore, we introduce an enhanced memory system enabling multiple concurrent memory accesses while remaining totally transparent to the C compiler.

The proposed design needs 15.8 ms for the computation of the Optimal-Ate pairing over a 256-bit BN curve at 338 MHz implemented with a 130 nm standard cell library. The processor core consumes 97 kGates making it suitable for the use in embedded systems.

Index Terms

Application-specific instruction-set processor (ASIP), design-space exploration, pairing-based cryptography, Barreto-Naehrig curves, elliptic-curve cryptography (ECC), $\mathbb{F}_p$-arithmetic.

This work has been supported by the UMIC Research Centre, RWTH Aachen University. The third author was supported by the European Commission through the ICT Programme under Contract ICT–2007–216499 CACE. Permanent ID of this document: 7e38974d56cc76a7f572f328ee4a3761. Date: Feb 5, 2009

1. Introduction

Pairings were first introduced to cryptography as a means to break cryptographic protocols based on the elliptic-curve discrete-logarithm problem (ECDLP) [1], [2]. Joux showed in 2000 that they can also be used constructively for tripartite key agreement [3]; other applications such as identity based encryption [4] and schemes to generate short digital signatures [5] have subsequently been introduced.

Cryptographic pairings are based on elliptic curves. To meet both, security requirements and computational feasibility, only elliptic curves with special properties can be considered as basis for cryptographic pairings. For the 128-bit security level, the best known curves are 256-bit Barreto-Naehrig curves (BN curves), introduced in [6]. Fast arithmetic on these curves demands for fast finite field arithmetic in a field $\mathbb{F}_p$ of prime order p, where p is determined by the curve construction. Additionally, the variety and complexity of pairing applications demand for a flexible and programmable solution. Application-specific instruction-set processors (ASIPs) are a promising candidate to find a good trade-off between these contradicting demands of speed, flexibility and ease of programmability.

This paper shows a design-space exploration of an ASIP for pairing computations over BN curves. The design does not target maximum speed at the cost of silicon area. Instead, we describe how to trade off execution time against area making the ASIP suitable for use in the embedded domain. Dedicated functional units are introduced that speed up general $\mathbb{F}_p$-arithmetic. Their critical path delay can be modified in order to be integrated with any existing RISC-like architecture without compromising its clock frequency. Independently from that adjustment, these units are scalable allowing for a trade-off between execution time and area consumption. The scaling influences the number of cycles required for a certain operation, but does not require any modification to the special instruction triggering it. Therefore, the same software can be used regardless of the selected size of the extensions. We show that the speedup from the special functional units is limited by a memory system with a single memory port. Hence, we introduce a memory system utilizing multiple memories. However, due to the resulting segmentation of the memory space this approach makes development of a C compiler a challenging task. As complexity of cryptographic pairings demands for a convenient programming model we address this issue by introducing an enhanced memory system which is totally transparent to the C compiler by hiding the memory space segmentation from the instruction set. The number of attached memories can thus be altered without changing the C compiler, the developed software or even the processor pipeline including the newly introduced special functional units. As a result, the proposed ASIP offers a flexible and scalable implementation for pairing applications.

We are—up to our knowledge—the first to implement and time a complete implementation of cryptographic pairings achieving a 128-bit security level on dedicated specialized hardware.

We would like to thank Jia Huang for supporting the implementation. We furthermore thank Daniel J. Bernstein, Tanja Lange, Ernst Martin Witte and Filippo Borlenghi for suggesting many improvements to our explanations.

Related work. Several architectures for the computation of cryptographic pairings have been proposed in the literature [7]–[19]. All these implemenations use supersingular curves over fields of characteristic 2 or 3. This choice, together with the choice of the underlying fields, yields security levels far below 128 bit. A comparative overview over these architectures is given in [7].

Barenghi et al. recently proposed a hardware architecture for cryptographic pairings using curves defined over fields of large prime characteristic [20]. They use a supersingular curve (with embedding degree 2) defined over a 512-bit field and thus achieve 80-bit security, according to [21].

Another architecture targeting speedup of pairings and supporing fields of large prime characteristic has been proposed in [22]. The instruction set of a SPARC V8 processor is extended for acceleration

of arithmetic in $\mathbb{F}_{2^n}$, $\mathbb{F}_{3^m}$ and $\mathbb{F}_p$. However, the focus is put on minor modifications of the datapath resulting in a performance gain for multiplications in $\mathbb{F}_p$ which is two-fold only. Our work focusses rather on significant datapath extensions in order to achieve high speedup for pairings in the embedded domain.

In [23], a special public-key cryptographic processor based on a SPARC CPU with special support for ECC point multiplication in $\mathbb{F}_p$ and $\mathbb{F}_{2^n}$ is presented. The operating frequency is 1.5 GHz targeting desktop general-purpose processors rather than the embedded domain.

The architectures closest to the one proposed in this paper are accelerating arithmetic in general $\mathbb{F}_p$ for elliptic-curve cryptography (ECC) [24]–[29]. However, these designs have not been reported to be used for complex applications like pairings. A detailed comparison with these architectures is given in Section 4.

Some other architectures for ECC over prime fields limit their support to a prime p which allows for particularly fast modular reduction (see i.e. [30]). These approaches are not adequate for pairing-based cryptography where additional properties of the elliptic curves are required. Thus, a detailed comparison with these architectures is omitted here.

Organization of the paper. Section 2 of the paper gives a short overview of cryptographic pairings and Barreto-Naehrig curves. Section 3 describes our approach of an ASIP suitable for pairing computation. In Section 4 we discuss the results. We futhermore give a comparison with specialized hardware targeting acceleration of elliptic-curve scalar multiplication on curves defined over fields of large prime characteristic described in the literature. The paper is concluded and future work is outlined in Section 5.

2. Background on cryptographic pairings

We only give a short overview of the notion of cryptographic pairings, a comprehensive introduction is given in [31, chapter IX].

For three groups G_1, G_2 (written additively) and G_3 (written multiplicatively) of prime order r a cryptographic pairing is a map $e : G_1 \times G_2 \rightarrow G_3$, with the following properties:

- Bilinearity:

  $e(kP, Q) = e(P, kQ) = e(P, Q)^k$ for $k \in \mathbb{Z}$.
- Non-degeneracy:

  For all nonzero $P \in G_1$ there exists $Q \in G_2$ such that $e(P, Q) \neq 1$ and
  for all nonzero $Q \in G_2$ there exists $P \in G_1$ such that $e(P, Q) \neq 1$.
- Computability:

  There exists an efficient algorithm to compute $e(P, Q)$ given P and Q.

We consider the following construction of cryptographic pairings: Let E be an elliptic curve defined over a finite field $\mathbb{F}_p$ of prime order. Let r be a prime dividing the group order $\#E(\mathbb{F}_p) = n = p + 1 - t$ and let k be the smallest integer, such that $r \mid p^k - 1$. We call k the embedding degree of E with respect to r.

Let $P_0 \in E(\mathbb{F}_p)$ and $Q_0 \in E(\mathbb{F}_{p^k})$ be points of order r such that $Q_0 \notin \langle P_0 \rangle$, let $\mathcal{O} \in E(\mathbb{F}_p)$ denote the point at infinity. Define $G_1 = \langle P_0 \rangle$ and $G_2 = \langle Q_0 \rangle$. Let $G_3 = \mu_r$ be the group of r-th roots of unity in $\mathbb{F}^*_{p^k}$.

For $i \in \mathbb{Z}$ and $P \in E$ a Miller function [32] is an element $f_{i,P}$ of the function field of E, such that the principal divisor of $f_{i,P}$ is $\mathrm{div}(f_{i,P}) = i(P) - ([i]P) - (i - 1)\mathcal{O}$.

Using such Miller functions, we can define the map

$$e_s : G_1 \times G_2 \rightarrow \mu_r; (P, Q) \mapsto f_{s,P}(Q)^{(p^k-1)/r}.$$

For certain choices of s the map e_s is non-degenerate and bilinear. For $s = r$ we obtain the reduced-Tate pairing τ and for $s = T = t - 1$ we obtain the reduced-Ate pairing α by switching the arguments [33]. Building on work presented in [34], Vercauteren introduced the Optimal-Ate pairing in [35] which for BN curves can be computed using $s \approx \sqrt{t}$ and a few additional computations (see also [36]).

Using twists of elliptic curves we can further define the generalized reduced-η pairing [33], [37]. In [38] a method to compute the Tate and η pairing keeping intermediate results in compressed form is introduced. We refer to the resulting algorithms as Compressed-Tate and Compressed-η pairing, respectively.

2.1. Choice of an Elliptic Curve

For cryptographic protocols to be secure on the one hand and the pairing computation to be computationally feasible on the other hand, the elliptic curve E must have certain properties: Security of cryptographic protocols based on pairings relies on the hardness of the discrete logarithm problem in G_1, G_2 and G_3. For the 128-bit security level, the National Institute of Standards and Technology (NIST) recommends a prime group order of 256 bit for $E(\mathbb{F}_p)$ and of 3072 bit for the finite field $\mathbb{F}_{p^k}$ [21].

Barreto-Naehrig curves, introduced in [6], are elliptic curves over fields of prime order p with embedding degree $k = 12$. The group order $n = r$ of $E(\mathbb{F}_p)$ is prime by construction, the values p and n can be given as polynomial expressions in an integer u as follows:

$$p = p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1 \text{ and}$$
$$n = n(u) = 36u^4 + 36u^3 + 18u^2 + 6u + 1.$$

For our implementation we follow [39] and set $u = \texttt{0x6000000000001F2D}$, yielding two primes $p(u)$ and $n(u)$ of $l = 256$ bit. The field size of $\mathbb{F}_{p^k}$ then has $256 \cdot k = 3072$ bit.

2.2. Computation of Pairings

The computation of cryptographic pairings consists of two main steps: the computation of $f_{s,P}(Q)$ for Tate and η pairings or of $f_{s,Q}(P)$ when considering the Ate pairing and the final exponentiation with $(p^k - 1)/r$.

The first part is usually done iteratively using variants of Miller's algorithm [32]. Several optimizations of this algorithm have been presented in [40]. The resulting algorithm is often referred to as BKLS algorithm. For BN curves even more optimizations can be applied by exploiting the fact that such curves have sextic twists. A detailed description of efficient computation of pairings over BN curves, including the computation of Miller functions and the final exponentiation is given in [39]. Our implementation follows this description in large parts.

Finite field computations constitute the bulk of the pairing computation – in software implementations typically more than 90% of the time is spent on modular multiplication, inversion, addition and subtraction. Throughout the pairing computation we keep points on elliptic curves in Jacobian coordinates and can thus almost entirely avoid field inversions.

Our targets for hardware acceleration are thus multiplication, addition and subtraction in $\mathbb{F}_p$.

3. An ASIP for Cryptographic Pairings

To implement various pairing algorithms, a programmable and therefore flexible architecture is targeted in this paper. Standard architectures like embedded RISC cores are flexible, but they are lacking sufficient computational performance for specific applications. Therefore, we apply the ASIP concept to cryptographic-pairing applications in order to reduce the computation time while maintaining programmability. Development and implementation of our ASIP have been carried out using the Processor

Designer from CoWare [41]. We used this tool suite for designing the actual architecture implementation on register transfer level (RTL) as well as the simulator and architecture specific software tools.

Keeping control over the data flow on the higher layers of the pairing computation, like $\mathbb{F}_{p^{12}}$ or $E(\mathbb{F}_{p^2})$ arithmetic, is a rather complex task. This calls for a convenient programming model. However, on the lower level realizing the $\mathbb{F}_p$-arithmetic, computational performance is of highest priority. Therefore, we decided to extend a basic RISC core with special $\mathbb{F}_p$ instructions. The available C compiler enables convenient application development on higher levels, while the computational intensive tasks are mapped to dedicated specialized instructions accessible via intrinsics[1]. The RISC core is a 32-bit five-stage pipelined load-store architecture featuring a 32-bit integer multiplier. $\mathbb{F}_p$ operations are the computationally expensive part of pairing computations. Table 1 shows the number of these operations for the each of the six implemented pairing applications. The low number of inversions does not justify the effort of a dedicated hardware implementation. We thus target modular multiplication as well as addition/subtraction for hardware based acceleration.

Application	*# mod mul*	*# mod add/sub*	*# inversions*
Optimal Ate	17,913	84,956	3
Ate	25,870	121,168	2
η	32,155	142,772	2
Tate	39,764	174,974	2
Compressed η	75,568	155,234	0
Compressed Tate	94,693	193,496	0

Table 1. Number of $\mathbb{F}_p$ operations for different pairing applications

Among those operations, the most challenging to implement is fast modular multiplication, especially for a large word width (e.g. 256 bit). In general, multiplication in $\mathbb{F}_p$ can be done by first multiplying the two factors (256 bit each) and then reducing the product (of 512 bit) modulo p. This might indeed be the fastest approach, if p could be chosen of a special form as for example specified in [21] or [42]. However, due to the construction of Barreto-Naehrig curves (see [6]) we cannot use such primes. Therefore, our approach uses Montgomery arithmetic [43].

The large word width also raises the issue of storing the data. A typical 16×32-bit RISC register file can store just two complete 256-bit words. Hence, not all required values can be kept locally in registers. In case of a simple load-store architecture additional instructions for memory accesses have to be executed which reduces the overall performance.

3.1. Data Processing: A Scalable Montgomery-Multiplier Unit

In 1985 Montgomery introduced an algorithm for modular multiplication of two integers A and B modulo an integer M [43]. The idea of the algorithm is to represent A as $\hat{A} = AR \mod M$ and B as $\hat{B} = BR \mod M$ for a fixed integer $R > M$ with $\gcd(R, M) = 1$. This representation is called Montgomery representation. To multiply two numbers in Montgomery representation we have to compute $\widehat{AB} = \hat{A}\hat{B}R^{-1} \mod M$. For certain choices of R this computation can be carried out much more efficiently than usual modular multiplication: Let us assume that M is odd and let l be the bitlength of M. Choosing $R = 2^l$ clearly fulfills the requirements on R and allows for modular multiplication that replaces division operations by shifts, allowing an efficient hardware implementation.

1. An adoption of our code to general purpose processors using the GMP library instead of intrinsics is available from http://cryptojedi.org/crypto/.

In the context of $\mathbb{F}_p$-multiplication the modulus M corresponds to p. All $\mathbb{F}_p$ operations can be performed in Montgomery representation. Therefore, all values can be kept in Montgomery representation throughout the whole pairing computation.

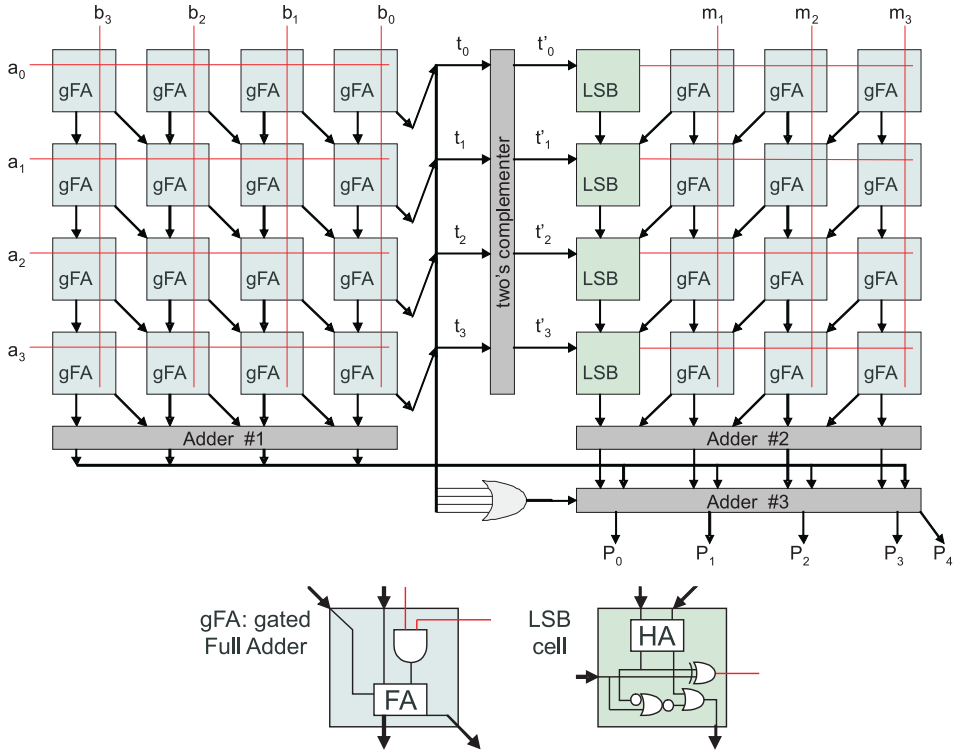

Figure 1. Montgomery-multiplier based on Nibouche et al. [44]

Nibouche et al. introduced a modified version of the Montgomery multiplication algorithm in [44]. It splits the algorithm into two multiplication operations, that can be carried out simultaneously, and allows using carry save (CS) multipliers. This results in a fast architecture that can be pipelined and segmented easily. Therefore, it is chosen as basis for our development. A 4×4-bit example is shown in Fig. 1.

The actual multiplication is carried out in the left half of the architecture, while the reduction is performed in the right part simultaneously. The left part is a conventional multiplier built of gated full adders (gFAs), whereas the right part consists of a multiplier with special cells for the least-significant bits (LSBs). The LSB cells are built around a half adder (HA). Their overall delay is comparable to that of a gFA. A more detailed description of the functionality can be found in [44].

Due to area constraints we decided to implement only subsets of the regular structures of the multiplier and perform the computation in multiple cycles. Intermediate results are stored in dedicated special registers. The CS-based design provides the opportunity to not only make horizontal but also vertical cuts. In our case, the horizontal cuts are made every eight gFAs. This height (H) of the multiplier implementation is selected such that the critical path delay of the unit is adjusted to that of the base RISC. Due to the CS-based design the critical path of the multiplier unit depends on its height only. This makes the design *adaptable* to existing cores in terms of *timing* maintaining the performance of their general instruction set. Once the height of the multiplier unit is chosen, the width (W) can be selected to *adapt* the design to the desired *computational performance* and to trade off *area vs. execution time* of the multiplication. The number and width of required registers to store intermediate data is independent

from this choice.

cycle	*partial multiplication ($W \times H$-bit)*	*partial reduction ($W \times H$-bit)*
$t_0 + 0$	$A[\ 7:\ 0] \times B[255:224]$	
$t_0 + 1$	$A[\ 7:\ 0] \times B[223:192]$	
$\vdots$	$\vdots$	
$t_0 + 7$	$A[\ 7:\ 0] \times B[\ 31:\ 0]$	
$t_0 + 8$	$A[\ 15:\ 8] \times B[255:224]$	$T'[\ 7:\ 0] \times M[255:224]$
$t_0 + 9$	$A[\ 15:\ 8] \times B[223:192]$	$T'[\ 7:\ 0] \times M[223:192]$
$\vdots$	$\vdots$	$\vdots$
$t_0 + 15$	$A[\ 15:\ 8] \times B[\ 31:\ 0]$	$T'[\ 7:\ 0] \times M[\ 31:\ 0]$
$\vdots$	$\vdots$	$\vdots$
$t_0 + 248$	$A[255:248] \times B[255:224]$	$T'[247:240] \times M[255:224]$
$t_0 + 249$	$A[255:248] \times B[223:192]$	$T'[247:240] \times M[223:192]$
$\vdots$	$\vdots$	$\vdots$
$t_0 + 255$	$A[255:248] \times B[\ 31:\ 0]$	$T'[247:240] \times M[\ 31:\ 0]$
$t_0 + 256$	addition #1	$T'[255:248] \times M[255:224]$
$t_0 + 257$		$T'[255:248] \times M[223:192]$
$\vdots$		$\vdots$
$t_0 + 263$		$T'[255:248] \times M[\ 31:\ 0]$
$t_0 + 264$		addition #2
$t_0 + 265$	addition #3	

Table 2. Time-flow example for 256-bit multi-cycle modular multiplication ($W = 32, H = 8$)

Multiplication and reduction are carried out simultaneously starting from the most-significant bit (MSB) of their second operand (B and M) first. However, the reduction cannot be started until the incoming data for the LSB cells are available from the two's complementer. Therefore, reduction starts after the first H lines of multiplication have been executed and remains delayed for $\lceil \frac{l}{W} \rceil$ cycles (required for the computation of H lines). Table 2 shows the resulting time flow for a 256-bit modular multiplication with a 32×8-bit unit. Eventually, the CS results need to be transformed back to two's complement number representation (by *addition #1* and *addition #2*) before they are combined to the result by *addition #3*. This is necessary since the result lies in the range of 0 to $2M - 1$, and requires a final comparison against M, which is difficult to handle in CS representation. The comparison including a necessary subtraction of M is performed in another functional unit introduced later. Equation (1) gives the number of required cycles c_{MM} to perform a Montgomery multiplication with the proposed multi-cycle architecture for the general case.

$$c_{MM} = \left(\left\lceil \frac{l}{H} \right\rceil + 1 \right) \cdot \left\lceil \frac{l}{W} \right\rceil + 2 \tag{1}$$

For evaluation, we implemented this Multi-cycle Montgomery-Multiplier (MMM) in three different sizes ($W \times H$): 32×8 bit, 64×8 bit and 128×8 bit, resulting in an execution time of 266, 134 and 68 cycles respectively. However, the area savings for smaller (and slower) architectures do not scale as well as the execution time. This results from the increased complexity of the required multiplexing for smaller MMM units. In order to keep the amount of multiplexers small, we designed special 256-bit shift

registers, that enable a circular shift by W bits for the operands B, M and the corresponding intermediate CS values. This solution is suitable, since the input values are accessed in consecutive order by blocks of W bits. Still, area savings when scaling a 128×8-bit architecture down to 32×8-bit are about 50%.

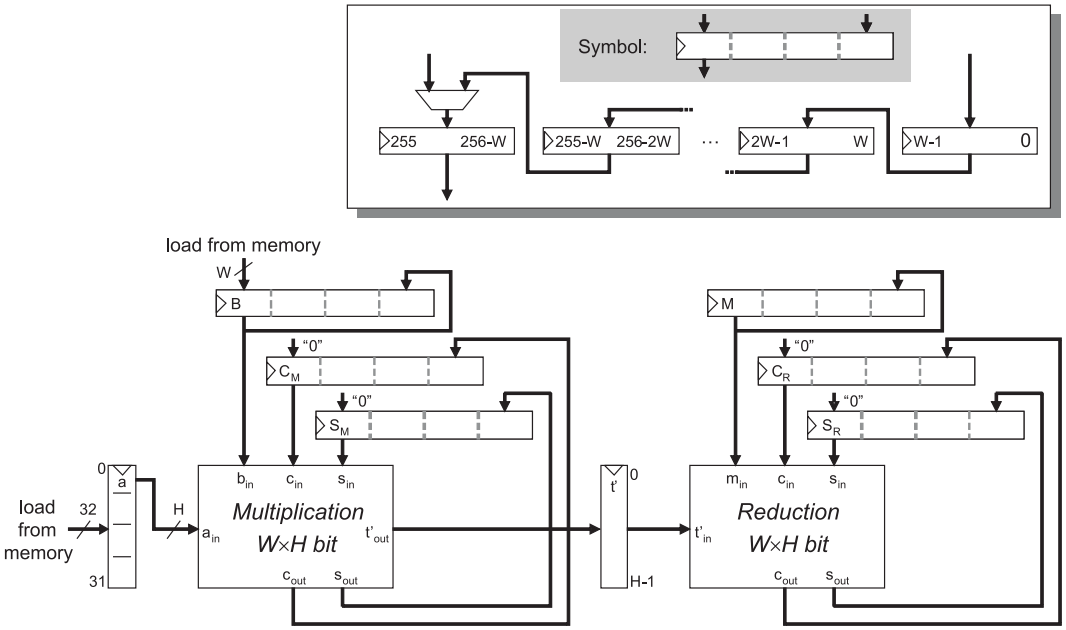

Figure 2. Structure of the multi-cycle Montgomery-multiplier (MMM)

Fig. 2 shows the overall resulting structure of the MMM unit. The two's complementer is included in the *multiplication unit*, while the *reduction unit* contains additional LSB cells that produce input for the gFA cells on the fly (as depicted in Fig. 1). The input shift registers are initialized step-by-step during the first $\lceil \frac{l}{W} \rceil$ cycles. After the whole process, the result is stored in the registers for temporary CS values (C_M, S_M, C_R, S_R). The adders for the final summations are not depicted.

An advantage of stepwise executing the multiplication is that the total multiplication width l can be configured at runtime in steps of W. The overall dependence of the execution time on l is quadratic. Modular multiplication is thus significantly faster for smaller multiplication width. This may be interesting for ECC applications that do not require 128-bit security.

Similar to the MMM unit we developed a *multi-cycle adder unit* for modular additions and subtractions, which reads two input operands block-wise and simultaneously. For evaluation, a 32-bit and a 64-bit version of this unit have been implemented. Details are omitted here since the implementation is straightforward. Please note that the adder unit causes higher demand on the throughput of data than the MMM, since an addition can be performed within much shorter time.

Both, MMM and *adder unit* require a final subtraction of M whenever the result exceeds this prime number. A special *writeback unit* takes care of this subtraction right before writing back the data, operating block-wise in multiple cycles as well. This unit has been implemented with a width of 32, 64 and 128 bit.

All three special units for $\mathbb{F}_p$-arithmetic are placed in the execute stage of the processor pipeline. During the execution of multi-cycle operations for modular addition, subtraction and multiplication the pipeline is stalled. Three special instructions are implemented triggering these operations. Instruction arguments are registers containing the starting address of each of the three 256-bit operands. Since the modulus M is not changed during an application run, a special register is utilized and implicitly accessed by the instructions.

This register is initialized with p at the beginning of an application via another dedicated instruction.

3.2. Data Access: An Enhanced Memory Architecture

Special instructions with high computational performance can result in high throughput demands for the memory system. In case of the proposed ASIP, especially the modular addition/subtraction on 256-bit operands requires a throughput higher than one 32-bit word per cycle. The following two evident mechanisms to increase memory throughput for ASIP designs are not well suited here: First, using memories with multiple ports is costly. The number of ports is limited to two for SSRAMs and the required area is roughly doubled. Second, designing a dedicated system with several (often specialized) memories targets highest performance, but is a complex task. The data memory space gets segmented irregularly, making it difficult to access and manage for a compiler.

Due to the drawbacks of these two approaches we apply a different technique, which we would like to introduce as *transparent interleaved memory segmentation (TIMS)*. Its basic principle is to extend the number of ports to the memory system in order to increase the throughput. TIMS splits the memory into regular blocks, which can be selected on the basis of address bits and accessed in parallel. In case of our ASIP, the LSBs of the address are used for the memory block selection. This results in an addressing scheme, where the memory block is selected by calculating the address modulo the number of memories m_d, which has to be a power of two. Because of its fairly simple mechanisms and regularity, the distribution of accesses to the memory system can be handled efficiently at runtime by a dedicated hardware block, the *memory-access unit (MAU)* (Fig. 3).

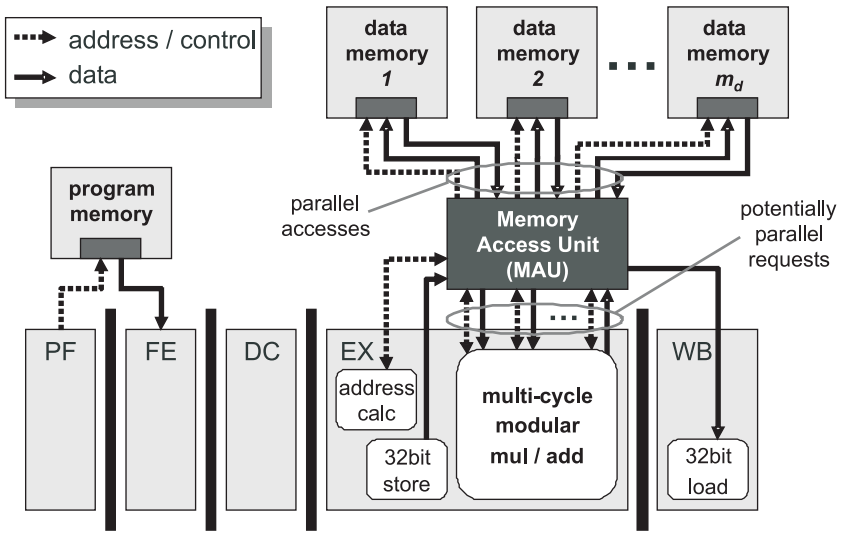

Figure 3. TIMS implementation with MAU

Memory accesses are requested concurrently by the pipeline on demand resulting in multiple independent read or write connections (unidirectional) between pipeline and MAU. The MAU takes care of distributing these requests and granting accesses. Therefore, a simple handshaking protocol is used between pipeline and MAU, which is able to confirm a request within the same cycle in order not to cause any delay cycles when trying to access the fast SSRAMs.

One advantage of this mechanism is the fact, that from the perspective of the core, the memory space remains unchanged, regardless of the number of attached memories. Existing load and store instructions are sufficient to access the whole data memory space. Even when special instructions perform concurrent

memory accesses, a modification in the memory system (e.g. changing number of attached memories) does not result in a change of the core, if the pipeline is designed properly. This enables orthogonal implementation and modification of the base architecture and the memory system.

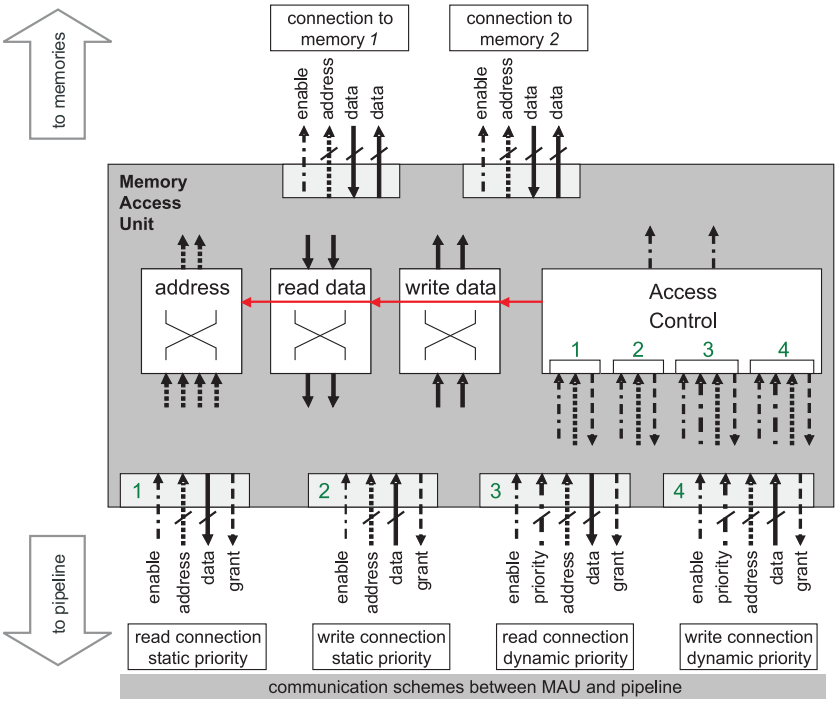

Figure 4. Interconnect of memory-access unit (MAU)

Of course, memory-access conflicts can occur when simultaneous accesses refer to the same memory. Therefore, a priority based resolution of access conflicts is implemented in the MAU in two ways. Static priorities can be used if certain accesses always have higher priority than others. For instance write accesses from later pipeline stages should always have higher priority than read accesses from prior stages. When the priority is changing at runtime, dynamic priority management is required. Then, dedicated additional communication lines between core and MAU indicate a change of priority. In our design this is required for the accesses from the *adder unit*.

Fig. 4 depicts the four different connection schemes between MAU and pipeline. The number and type of connections between MAU and pipeline are determined by the number and type of independent memory accesses initiated by the pipeline, while the number of actual memory connections depends on the number of attached data memories m_d ($m_d = 2$ in this example). For sake of clarity, the actual interconnections within the MAU have been omitted in Fig. 4. The *access-control* block combines the *enable* and *priority* signals with the $\log_2(m_d)$ LSBs of the *address* signals from the read and write connections in order to produce the *grants*. At the same time the *enable* signals for the SSRAMs are set accordingly by this unit. It also controls the crossbars that are switching the correct *address*, *read data* and *write data* signals to the memory ports. Please note, that the *read data* crossbar is switched with one cycle delay compared to the *address* crossbar in order to be in sync with the single cycle read latency of the SSRAMs. Effects of TIMS on physical parameters like timing and area consumption are discussed in detail in the result section.

Memory access collisions decrease the performance of the system and cannot be avoided completely due to the automatic address management of the C compiler. However, in our case this effect is kept

minimal due to the good distribution of the 256-bit words. For additions and multiplications this causes a maximum additional delay of one single cycle only. This results in a maximum performance degradation caused by memory-access conflicts of less than 2% for any of the implemented pairing applications.

	# data memories (single 32-bit port)		
256-bit $\mathbb{F}_p$-Operation	4 *(128-bit writeback)*	2 *(64-bit writeback)*	1 *(32-bit writeback)*
mod mul (128×8-bit unit)	79	83	91
mod mul (64×8-bit unit)	150 - 151	153	161
mod mul (32×8-bit unit)	294 - 295	296 - 297	301
mod add (64-bit unit)	8 - 9	14 - 15	26
mod add (32-bit unit)	12 - 13	14 - 15	26

Table 3. Number of cycles for 256-bit $\mathbb{F}_p$-arithmetic including memory access

Due to hidden latencies of memory accesses and possible memory-access collisions, the execution times of the arithmetic functional units and the memory system cannot be subsumed in a simple equation. Table 3 presents the cycle count for each operation depending on the number of attached memories. It can be seen that the cycle count for an addition using the 64-bit *adder unit* does not differ from the case utilizing a 32-bit unit, when one or two 32-bit wide memories are attached. The throughput of the memory system limits the maximum achievable performance in this case. Not adapting the width of the adder to the memory system therefore wastes either performance or area, this also holds for the *writeback unit*. Thus the width of the *writeback unit* is coupled to the number of memories in the table. Note that the performance gain of the 32-bit *adder unit* in the four-memories case results from a faster writeback and not from an actual speedup of the addition. The implementation of a 16-bit adder for the single-memory case would not reduce area significantly due to additional multiplexing and is therefore neglected. The width of the multiplier can be selected independently from the number of memories, since operands do not need to be fetched continuously from the memory and smaller memory bandwidth reduces performance only slightly.

4. Results

Overall, we have implemented nine variants of our ASIP with different design parameters regarding number of data memories and width of the computational units for modular multiplication, modular addition and multi-cycle writeback (Table 4). As explained in the previous section, the number of data memories is closely coupled with the width of the *adder* and the *writeback unit*. All synthesis results have been obtained with Synopsys Design Compiler [45] using a 130 nm CMOS standard cell library with a supply voltage of 1.2 V. The memories are synchronous single port SRAMs with a latency of one cycle. The total data memory size is 2048 words for each of the design variants, while the number of memories can vary according to TIMS. The program memory is not included in the area reports, since it is not changing through the different designs and could be implemented differently (as ROM, RAM, synthesized logic etc.) depending on the final target system.

Fig. 5 shows the area distribution of the different ASIP variants. While the basic core only shows moderate area increase from 17 to 21 kGates for all variants (resulting from decoder extensions and additional pipeline registers), the area for the register file increases from 9 to 28 kGates compared to the plain RISC. The reason are specialized 256-bit registers storing the prime number and intermediate results of the modular operations. These registers are independent from the width of any of the additional

Variant	*128m4*	*64m4*	*32m4*	*128m2*	*64m2*	*32m2*	*128m1*	*64m1*	*32m1*	*plain RISC*
mod mul size *(bit)*	128×8	64×8	32×8	128×8	64×8	32×8	128×8	64×8	32×8	-
mod add width *(bit)*	64	64	64	32	32	32	32	32	32	-
writeback width *(bit)*	128	128	128	64	64	64	32	32	32	-
# data memories	4	4	4	2	2	2	1	1	1	1
total area[2] *(kGates)*	195	186	182	164	153	148	145	134	130	77
core area[3] *(kGates)*	96	87	83	97	86	81	93	83	79	26
timing *(ns)*	3.69	3.65	3.52	2.96	2.97	3.02	2.95	3.03	3.09	2.89
Optimal Ate *(ms)*	17.5	21.8	29.9	15.8	19.4	27.3	19.2	23.4	32.0	-
Ate *(ms)*	25.3	31.4	42.6	22.8	27.9	38.9	27.6	33.5	45.6	-
η *(ms)*	32.3	39.5	52.8	28.8	35.0	48.1	34.6	41.6	56.2	-
Tate *(ms)*	38.5	47.0	62.7	34.4	41.6	57.1	41.1	49.5	65.3	-
Compressed η *(ms)*	38.6	55.0	86.2	34.5	48.2	77.1	41.6	56.5	85.8	-
Compressed Tate *(ms)*	48.2	68.9	107.8	43.2	60.3	96.5	52.0	70.7	107.3	-

Table 4. Implemented design variants of the ASIP for pairings

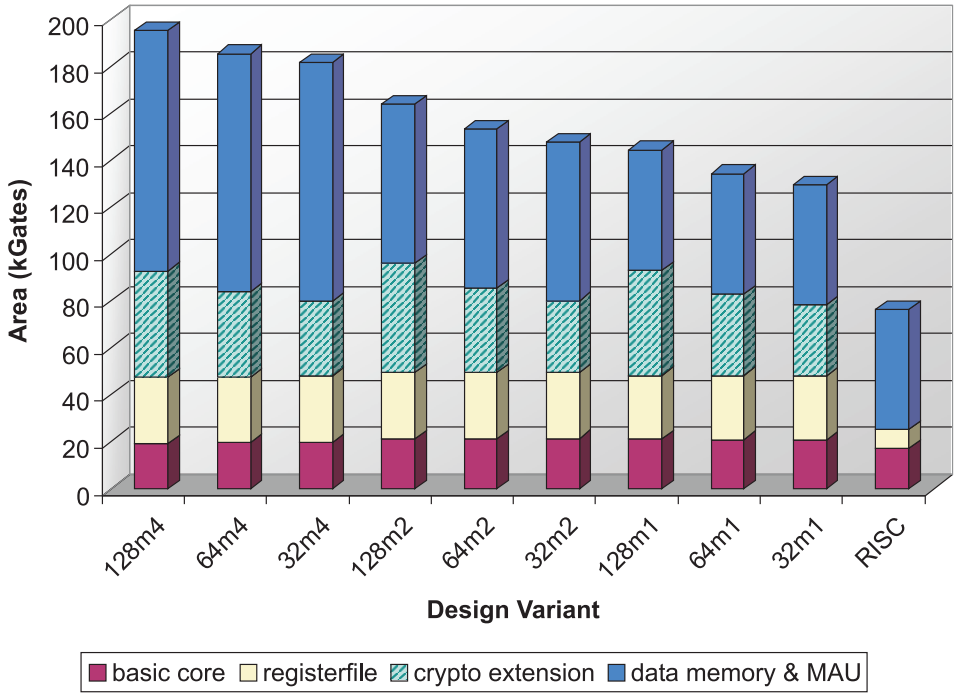

Figure 5. ASIP area consumption and distribution

functional units. Therefore, the register file area is equal for all variants. The area of the cryptographic extensions is dominated by the MMM unit.

Observe that splitting the memory into two of half the size results in a data-memory area increase of 31%. Utilizing a dual port memory instead would increase area by over 83%. Splitting the memory into

2. Including area for data memories
3. Without area for memories, but including area for MAU

four parts causes an area overhead of 94%, but also enables accessing four words at the same time, as long as there are no conflicts. The area overhead due to the MAU lies between only 0.5 and 1.2 kGates, when two memories are attached. Even for four attached memories it is below 3.5 kGates.

However, limitations of TIMS utilizing the proposed MAU become visible when looking at the timing of the different variants of the ASIP. While attaching one or two data memories barely affects the critical path with respect to the original RISC architecture (within design tool accuracy, see Table 4), an additional delay is observed when four memories are attached. This delay is caused by the complexity of priority resolution for four attached memories combined with four independent memory accesses with dynamic priority, which are necessary to implement the 64-bit adder. However, maximum frequencies of ASIPs are often designed to be lower than in this case. For our design it is possible (and reasonable) to maintain the clock frequency of the original RISC although the instruction set is extended with quite complex instructions, due to the adjustable delay of the MMM.

The execution times of all six implemented pairing applications on all nine ASIP variants are shown in Table 4. Note that running an application on different ASIP variants does not require recompling the application because the instruction set is identical for all of them. For all applications performance improves significantly with increasing width of the MMM. Also, the number of cycles decreases when increasing the number of connected data memories. Unfortunately, the longer critical path of the four-memory system leads to a lower performance than for the designs with two memories. The overall fastest design is variant *128m2*, executing the Optimal-Ate pairing in 15.8 ms. With the smallest and slowest variant completing the task in 32.0 ms, the user is offered a quite broad design space enabling trade-offs.

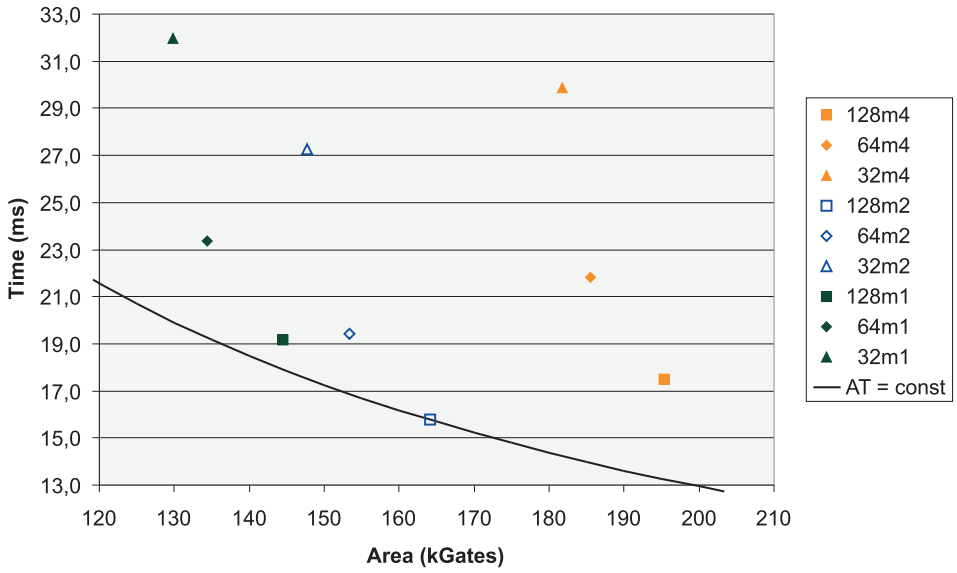

Figure 6. Area-time trade-off for different ASIP variants and the Optimal-Ate-pairing application

In order to evaluate the efficiency of the different design variants, Fig. 6 shows the area-time trade-off for the Optimal-Ate pairing. It can be seen clearly that the best AT product is obtained by the *128m2* design. This shows the importance of investigaing the memory architecture of ASIPs during design-space exploration. In our case the best results are obtained with TIMS and *two* data memories in spite of the considerable area increase due to the memory splitting.

4.1. Performance Comparison

There exists no literature reporting performance figures resulting from actual implementations of cryptographic pairings on dedicated hardware achieving a 128-bit security level. Hardware implemenatations for lower security levels and implementations using more area can obviously be much faster than the proposed design. For example, Beuchat et al. in [7] report a computation time of 24 μs for the η_T pairing on a supersingular 154-bit curve using 11318 slices of a Xilinx xc4vlx25 FPGA running at 200 MHz. An even faster implementation of the same pairing is reported in [16]. It uses 74105 slices of a Xilinx xc4vlx200 FPGA running at 199.227 MHz to compute the η_T pairing in 8.17 μs. Observe that this setting does not even achieve 80-bit security level.

In [39], Devigili et al. report 5.17 s for the computation of the Ate pairing over a 256-bit Barreto-Naehrig curve on a Philips HiPerSmart™ smart card operating at 20.57 MHz. This smart card contains a SmartMIPS-based 32-bit architecture. The paper thus gives an impression of the achievable performance for the computation of cryptographic pairings in the embedded domain without highly specialized hardware.

Other publications describing hardware for ECC over fields of large prime characteristic give performance figures in terms of time needed for a scalar multiplication with a scalar k of a certain size, e.g. the computation of $[k]P$ for some $P \in E(\mathbb{F}_p)$.

The first dedicated hardware implementation of arithmetic on elliptic curves over $\mathbb{F}_p$ is presented in [24]. The authors estimate 3 ms for a scalar multiplication with a 192-bit scalar on a curve over a 192-bit field for an implementation on a Xilinx XCV1000E-8-BG680 (Virtex E) field-programmable gate array (FPGA) operating at 40 MHz. However, this estimate assumes a 100% utilization of the multiplier.

Another FPGA implementation of ECC accelerating general $\mathbb{F}_p$-arithmetic is described in [26]. This implementation needs less memory and can achieve higher clock frequencies. It features a linear systolic array to speed up Montgomery modular multiplication. This approach leads to a high throughput and massively parallel computation, since the multiplications in $\mathbb{F}_p$ are distributed bitwise over multiple Processing Elements (PEs). For the proposed design with 160 PEs, scalar multiplication with a 160-bit scalar on an elliptic curve over a 160-bit field is reported to take 14.4 ms at a clock frequency of 91.308 MHz on a Xilinx V1000E-BG-560-8 (Virtex E) FPGA.

The processor presented in [27] is the fastest FPGA implementation of ECC over $\mathbb{F}_p$ reported to date. A 256-bit scalar multiplication is executed in 3.86 ms at 39.5 MHz on a Xilinx XC2VP125-7-ff1696 (Virtex2 Pro). The design features a full 256-bit multiplier utilizing embedded multipliers and the fast carry look-ahead logic of the FPGA. As our ASIP is designed for embedded systems, a 256×256-bit multiplier is clearly not suitable due to area constraints.

A direct comparison of this work with these FPGA implementations in terms of area and speed is difficult, because our design is implemented using a 130 nm standard cell library. However, there are two publications targeting ASIC implementations on 130 nm as well, which are discussed in the following two paragraphs.

In [25] a processor is presented that speeds up ECC calculations with a dual-field multiplier, which supports multiplications in $\mathbb{F}_{2^n}$ and $\mathbb{F}_p$. The architecture is scalable and depending on the multiplier size different speedups can be achieved at the cost of increased area. However, the multiplier is restricted to quadratic sizes of $r \times r$ bit. The size of the multiplier therefore has a huge impact on the critical path of the design and large multipliers decrease the maximum clock frequency significantly. With a 64×64-bit multiplier and implemented with a 130 nm CMOS standard cell library, the design requires 107 kGates (without memories) and performs a 256-bit scalar multiplication in $\mathbb{F}_p$ at a maximum achievable clock frequency of 138 MHz in 2.68 ms.

Similar to the work in [26], [28] presents an ECC processor based on a systolic arithmetic unit, which is extended to also support binary fields in [29]. $\mathbb{F}_p$ operations (including multiplication and division)

are executed on a unified systolic array with 130 PEs. The pipelined structure leads to highly parallel computation at high frequencies. However, one issue with this technique is the utilization of this structure. Conditions and data dependencies can lead to pipeline stalls decreasing the overall performance of the system. The authors reduce this effect by instruction reordering and utilization of delay slots. Another drawback is that the data-transfer mode of the datapath is of bit granularity and high concurrency, while the memory access is of word granularity and low concurrency. The authors therefore introduce latching registers close to the PEs in order to access the data in time and place. This comes at a cost of increased area and critical-path delay. The presented processor can operate at a frequency of 556 MHz and consumes 122 kGates. It performs a 256-bit scalar multiplication in 1.01 ms. Like our design it is implemented with a 130 nm standard cell library. For elliptic-curve arithmetic it uses points in affine representation; finite field inversions are accelerated through hardware.

In order to compare the results of this work with these architectures we implemented scalar multiplication on the 256-bit Barreto-Naehrig curve that we also used for pairing computation. Our design does not accelerate field inversion through hardware, so we use Jacobian projective coordinates to represent the points on the curve, trading inversions for several multiplications.

We emphasize that the choice of this curve is far from being optimal in terms of achievable performance for ECC not involving pairings. Clearly using an elliptic curve in Edwards form [46] and representing points in inverted Edwards coordinates [47] would improve speed for scalar multiplication significantly.

As in [25] and [28], we use the so-called NAF method for scalar multiplication: The scalar is first transformed into non-adjacent form (NAF); scalar multiplication is then carried out scanning this representation from left to right (see e.g. [48, Section 3.3.1]). For each signed-bit entry of the NAF one point doubling is performed; if the entry is non-zero an additional point addition is performed. On average only $\frac{1}{3}$ and at most $\frac{1}{2}$ of the signed bits of a NAF are non-zero. Clearly the number of non-zero entries of the NAF affects the computation speed of a scalar multiplication. For benchmarking we follow [25] and assume a scalar with $\frac{1}{3}$ non-zero entries.

Scalar multiplication with a 256-bit scalar takes 0.998 ms for the *128m2* variant of the proposed design at a maximum achievable clock frequency of 338 MHz. This number includes transformation of the scalar into NAF and a transformation from Jacobian into affine coordinates at the end.

Note that ASIP variant *128m2* is not only slightly faster than the designs in [25] and [28], but also consumes less area (97 kGates). However, it should be noted that the utilization of two data memories in our design affects on the overall area consumption.

5. Conclusion and Outlook

In this paper we presented a design-space exploration of an ASIP for computation of cryptographic pairings over BN curves. The design is based on extensions of an existing RISC core, which are completely transparent and independent from the original pipeline. Therefore, they could be applied to any RISC-like architecture, which can stall the pipeline during multi-cycle operations. The extensions are adaptable in terms of timing and allow for a trade-off between execution time and area. A flexible and transparent memory architecture extension making use of multiple memories (TIMS) enables the usage of existing compilers, since the address space remains unsegmented.

Future objectives are including countermeasures against side-channel attacks, which are not specially targeted in the current design, either in hard- or in software.

References

[1] A. Menezes, T. Okamoto, and S. Vanstone, “Reducing elliptic curve logarithms to logarithms in a finite field,” *IEEE Trans. Information Theory*, vol. 39, no. 5, pp. 1639–1646, 1993.

[2] G. Frey and H.-G. Rück, "A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves," *Math. of Computation*, vol. 62, no. 206, pp. 865–874, 1994.

[3] A. Joux, "A one round protocol for tripartite Diffie-Hellman," in *Algorithmic Number Theory*, ser. LNCS, vol. 1838, 2000, pp. 385–394.

[4] D. Boneh and M. Franklin, "Identity based encryption from the Weil pairing," in *Advances in Cryptology – CRYPTO 2001*, ser. LNCS, vol. 2139, 2001, pp. 213–229.

[5] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," *J. Cryptology*, vol. 17, no. 4, pp. 297–319, 2004.

[6] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *Selected Areas in Cryptography*, ser. LNCS, vol. 3897, 2006, pp. 319–331.

[7] J.-L. Beuchat, N. Brisebarre, J. Detrey, E. Okamoto, and F. Rodríguez-Henríquez, "A comparison between hardware accelerators for the modified Tate pairing over $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$," in *Pairing-Based Cryptography – Pairing 2008*, ser. LNCS, vol. 5209, 2008, pp. 297–315.

[8] J.-L. Beuchat, N. Brisebarre, J. Detrey, E. Okamoto, M. Shirase, and T. Takagi, "Algorithms and arithmetic operators for computing the η_t pairing in characteristic three," *IEEE Trans. Comput.*, vol. 57, no. 11, pp. 1454–1468, 2008.

[9] J.-L. Beuchat, M. Shirase, T. Takagi, and E. Okamoto, "An algorithm for the η_t pairing calculation in characteristic three and its hardware implementation," in *Proc. 18th IEEE Symp. Computer Arithmetic*, 2007, pp. 97–104.

[10] J.-L. Beuchat, H. Doi, K. Fujita, A. Inomata, A. Kanaoka, M. Katouno, M. Mambo, E. Okamoto, T. Okamoto, T. Shiga, M. Shirase, R. Soga, T. Takagi, A. Vithanage, and H. Yamamoto, "FPGA and ASIC implementations of the η_t pairing in characteristic three," Cryptology ePrint Archive, Report 2008/280, 2008, http://eprint.iacr.org/2008/280.

[11] C. Shu, S. Kwon, and K. Gaj, "FPGA accelerated Tate pairing based cryptosystems over binary fields," in *Proc. IEEE Int'l Conf. Field Programmable Technology – FPT 2006*, 2006, pp. 173–180.

[12] M. Keller, R. Ronan, W. Marnane, and C. Murphy, "Hardware architectures for the Tate pairing over GF(2^m)," *Computers & Electrical Eng.*, vol. 33, no. 5-6, pp. 392–406, 2007.

[13] M. Keller, T. Kerins, F. Crowe, and W. Marnane, "FPGA implementation of a GF(2^m) Tate pairing architecture," in *Reconfigurable Computing: Architectures and Applications*, ser. LNCS, vol. 3985, 2006, pp. 358–369.

[14] R. Ronan, C. Ó hÉigeartaigh, C. Murphy, M. Scott, and T. Kerins, "FPGA acceleration of the Tate pairing in characteristic 2," in *Proc. IEEE Int'l Conf. Field Programmable Technology*, 2006, pp. 213–220.

[15] P. Grabher and D. Page, "Hardware acceleration of the Tate pairing in characteristic three," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, ser. LNCS, vol. 3659, 2005, pp. 398–411.

[16] J. Jiang, "Bilinear pairing (Eta_T pairing) IP core," Tech. Rep., 2007, http://www.cs.cityu.edu.hk/~ecc/doc/etat_datasheet_v2.pdf.

[17] T. Kerins, W. P. Marnane, E. M. Popovici, and P. S. L. M. Barreto, "Efficient hardware for the Tate pairing calculation in characteristic three," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, ser. LNCS, vol. 3659, 2005, pp. 412–426.

[18] R. Ronan, C. Murphy, T. Kerins, C. Ó hÉigeartaigh, and P. Barreto, "A flexible processor for the characteristic 3 η_t pairing," *Int'l J. High Performance Systems Architecture*, vol. 1, no. 2, pp. 79–88, 2007.

[19] G. Kömürcü and E. Savas, "An efficient hardware implementation of the Tate pairing in characteristic three," in *Proc. Third Int'l Conf. Systems – ICONS 2008*, 2008, pp. 23–28.

[20] A. Barenghi, G. Bertoni, L. Breveglieri, and G. Pelosi, "A FPGA coprocessor for the cryptographic Tate pairing over $\mathbb{F}_p$," in *Proc. Fifth Int'l Conf. Information Technology: New Generations – ITNG 2008*, 2008, pp. 112–119.

[21] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management - part 1: General (revised)," National Institute of Standards and Technology, NIST Special Publication 800-57, 2007. http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf

[22] T. Vejda, D. Page, and J. Großschädl, "Instruction set extensions for pairing-based cryptography," in *Pairing-Based Cryptography – Pairing 2007*, ser. LNCS, vol. 4575, 2007, pp. 208–224.

[23] H. Eberle, S. Shantz, V. Gupta, N. Gura, L. Rarick, and L. Spracklen, "Accelerating next-generation public-key cryptosystems on general-purpose CPUs," *IEEE Micro*, vol. 25, no. 2, pp. 52–59, 2005.

[24] G. Orlando and C. Paar, "A scalable GF(p) elliptic curve processor architecture for programmable hardware," in *Cryptographic Hardware and Embedded Systems – CHES 2001*, ser. LNCS, vol. 2162, 2001, pp. 348–363.

[25] A. Satoh and K. Takano, "A scalable dual-field elliptic curve cryptographic processor," *IEEE Trans. Computers*, vol. 52, no. 4, pp. 449–460, 2003.

[26] S. Örs, L. Batina, B. Preneel, and J. Vandewalle, "Hardware implementation of an elliptic curve processor over GF(p)," in *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors – ASAP 2003*, 2003, pp. 433–443.

[27] C. McIvor, M. McLoone, and J. McCanny, "Hardware elliptic curve cryptographic processor over GF(p)," *IEEE Trans. Circuits and Systems I: Regular Papers*, vol. 53, no. 9, pp. 1946–1957, 2006.

[28] G. Chen, G. Bai, and H. Chen, "A high-performance elliptic curve cryptographic processor for general curves over GF(p) based on a systolic arithmetic unit," *IEEE Trans. Circuits and Systems II: Express Briefs*, vol. 54, no. 5, pp. 412–416, 2007.

[29] G. Chen, G. Bai, and H. Chen, "A dual-field elliptic curve cryptographic processor based on a systolic arithmetic unit," in *Proc. IEEE Int'l Symp. Circuits and Systems – ISCAS 2008*, 2008, pp. 3298–3301.

[30] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Cryptographic Hardware and Embedded Systems – CHES 2008*, ser. LNCS, vol. 5154, 2008, pp. 62–78.

[31] S. Galbraith, "Pairings," in *Advances in Elliptic Curve Cryptography*, ser. London Mathematical Society Lecture Note Series, I. F. Blake, G. Seroussi, and N. P. Smart, Eds. Cambridge University Press, 2005, ch. IX.

[32] V. S. Miller, "The Weil pairing, and its efficient calculation," *J. Cryptology*, vol. 17, pp. 235–261, 2004.

[33] F. Hess, N. Smart, and F. Vercauteren, "The Eta pairing revisited," *IEEE Trans. Information Theory*, vol. 52, no. 10, pp. 4595–4602, 2006.

[34] E. Lee, H.-S. Lee, and C.-M. Park, "Efficient and generalized pairing computation on Abelian varieties," Cryptology ePrint Archive, Report 2008/040, 2008. http://eprint.iacr.org/2008/040

[35] F. Vercauteren, “Optimal pairings,” Cryptology ePrint Archive, Report 2008/096, 2008. http://eprint.iacr.org/2008/096

[36] F. Hess, “Pairing lattices,” in *Pairing-Based Cryptography – Pairing 2008*, ser. LNCS, vol. 5209, 2008, pp. 18–38.

[37] P. S. L. M. Barreto, S. D. Galbraith, C. Ó hÉigeartaigh, and M. Scott, “Efficient pairing computation on supersingular Abelian varieties,” *Designs, Codes and Cryptography*, vol. 42, no. 3, pp. 239–271, 2007.

[38] M. Naehrig, P. S. L. M. Barreto, and P. Schwabe, “On compressible pairings and their computation,” in *Progress in Cryptology – AFRICACRYPT 2008*, ser. LNCS, vol. 5023, 2008, pp. 371–388.

[39] A. J. Devegili, M. Scott, and R. Dahab, “Implementing cryptographic pairings over Barreto-Naehrig curves,” in *Pairing-Based Cryptography – Pairing 2007*, ser. LNCS, vol. 4575, 2007, pp. 197–207.

[40] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott, “Efficient algorithms for pairing-based cryptosystems,” in *Advances in Cryptology – CRYPTO 2002*, 2002, pp. 354–368.

[41] CoWare Processor Designer. http://www.coware.com/products/processordesigner.php

[42] D. J. Bernstein, “Curve25519: new Diffie-Hellman speed records,” in *Public Key Cryptography – PKC 2006*, ser. LNCS, vol. 3386, 2006, pp. 207–228.

[43] P. Montgomery, “Modular multiplication without trial division,” *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[44] O. Nibouche, A. Bouridane, and M. Nibouche, “Architectures for Montgomery’s multiplication,” *IEE Proc. – Computers and Digital Techniques*, vol. 150, no. 6, pp. 361–368, 2003.

[45] Synopsys Design Compiler. http://www.synopsys.com/products/logic/design_compiler.html

[46] D. J. Bernstein and T. Lange, “Faster addition and doubling on elliptic curves,” in *Advances in Cryptology – ASIACRYPT 2007*, ser. LNCS, vol. 4833, 2007, pp. 29–50.

[47] D. J. Bernstein and T. Lange, “Inverted Edwards coordinates,” in *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, ser. LNCS, vol. 4851, 2007, pp. 20–27.

[48] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Secaucus, NJ, USA: Springer-Verlag, 2003.