Journal of Signal Processing Systems 51, 39–55, 2008
© 2007 Springer Science + Business Media, LLC. Manufactured in The United States.
DOI: 10.1007/s11265-007-0135-9

# Parallel Memory Architecture for Elliptic Curve Cryptography over $\mathbb{GF}(p)$ Aimed at Efficient FPGA Implementation

RALF LAUE AND SORIN A. HUSS
*Integrated Circuits and Systems Laboratory, Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany*

*Received: 14 April 2007; Revised: 18 July 2007; Accepted: 7 August 2007*

**Abstract.** Parallelization of operations is of utmost importance for efficient implementation of Public Key Cryptography algorithms. Starting with a classification of parallelization methods at different abstraction levels of public key algorithms, we propose a novel memory architecture for elliptic curve implementations with multiple modular multiplier units. This architecture is well-suited for different point addition and doubling algorithms over $\mathbb{GF}(p)$ to be implemented on FPGAs. It allows the execution time to scale with the number of modular multipliers and exhibits nearly no overhead compared to the mere runtime of the multipliers. The advantages of this distributed memory architecture are demonstrated by means of two different point addition and doubling algorithms.

**Keywords:** elliptic curve cryptography, parallelization, memory architecture, FPGA

## 1. Introduction

Today's hardware is becoming increasingly parallel: Modern workstation processors feature Single Instruction Multiple Data (SIMD) functionality, FPGAs contain massive amounts of identical simple reconfigurable components, and System-on-a-Chip (SoC) promises to be the major architecture for embedded systems in the future. Cryptographic implementations (restricted to public key cryptography in this context) should take advantage of this inherent concurrency by using parallel algorithm variants.

Recent years saw a rising in the amount of proposals to utilize these parallel hardware structures for more efficient implementations of public key cryptography—particularly RSA and Elliptic Curve Cryptography (ECC). Most approaches concentrate on parallelization within the modular arithmetic, especially the modular multiplication. But there are also proposals aiming at higher abstraction levels, in particular for FPGAs and SIMD processors.

After providing an overview on parallelization approaches on different abstraction levels, we propose a novel parallel memory architecture for FPGAs, mainly applicable to ECC on $\mathbb{GF}(p)$. The architecture scales well in relation to the number of modular multipliers by employing the distributed memory resources in parallel, while keeping resource demands low by constraining the usage of register and routing resources.

The following section offers a systematic overview on parallelization methods for common public key algorithms. Section 3 reviews properties of designs with parallel arithmetical units and details the new architecture. Section 4 introduces the method employed in this work for scheduling the modular operations on the parallel hardware resources. Then it applies this method to the IEEE point addition and doubling algorithms and, subsequently, presents the

This work was supported by the research program SicAri of BMBF (German Federal Ministry of Education and Research)

results of a prototype implementation and compares it with related realizations. Furthermore, it contains an example showing the figures of merit of the proposed architecture for a completely different EC algorithm. The last section, finally, concludes with some closing remarks.

## 2. Parallelization on Different Abstraction Levels

This section presents a systematic overview on generic approaches for hardware parallelization on different abstraction levels of the common public key protocols RSA and ECC as depicted in Fig. 1. The Modular Arithmetic constitutes the lowest abstraction level. It contains the modular operations like multiplication, addition, and subtraction, which are employed by higher abstraction levels. For elliptic curves only, the next higher level is the *Elliptic Curve Group* level. For this the underlying modular arithmetic must be a finite field, i.e. for $\mathbb{GF}(p)$ the modulus $p$ must be a prime number. The third abstraction level is the Cryptographic Main Operation level. For elliptic curves, it comprises the Point Multiplication, which uses the additive operations of the underlying elliptic curve group. For RSA, the Exponentiation builds directly on the modular arithmetic using its properties as a multiplicative group,

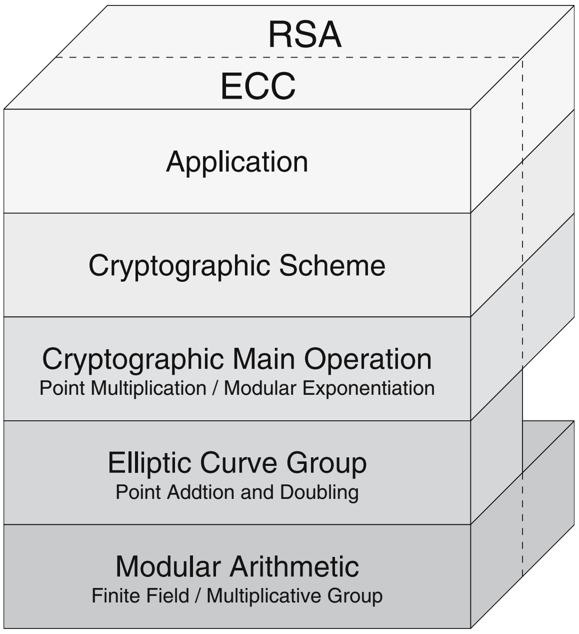
RSA
ECC
Application
Cryptographic Scheme
Cryptographic Main Operation
Point Multiplication / Modular Exponentiation
Elliptic Curve Group
Point Addtion and Doubling
Modular Arithmetic
Finite Field / Multiplicative Group

*Figure 1.* Abstraction levels of ECC and RSA algorithms.

i.e. the modular arithmetic is not a finite field and the modulus is not prime. The next abstraction level encapsulates the cryptographic main operation together with some auxiliary functions like hash function or random number generation to form a complete Cryptographic Scheme. For simplicity these auxiliary functions are not depicted in Fig. 1, as they are not within the scope of this work. The highest abstraction level is the Application level, which employs one or several cryptographic schemes for some meaningful application.

In general, parallelization yields the greatest benefit on lower abstraction levels, because control logic and storage within the parallelized modules increase considerably on higher levels. But as parallelization on higher levels allows a further speed-up and offers advantages not available on lower levels, it should be taken into consideration, too. Note that the following methods do not exclude each other, they can easily be combined.

### *2.1. Parallelization on Modular Arithmetic Level*

The modular arithmetical units are composed of elementary modules operating on data chunks with sizes ranging from bits to words. Since the modular multiplication is the most important operation, it is a popular strategy to run its internal units in parallel. In some cases this also implies parallel structures for the modular arithmetic besides multiplication (see below: buses of full bit-width and RNS). Parallelization within modular operations except multiplication is generally not required, as those operations are much faster than multiplication. The presented methods do not exclude each other and constitute no exhaustive list, although they seem to be the most common.

*Data-paths of full bit-width* Multiplication generally has a quadratic complexity relative to the bit-width. The execution time can be reduced to linear complexity by considering all bits at once. Of course, this comes at the cost of an approximate proportional increase in terms of resources. The systolic array implementing a modular multiplication from [1], for example, has about twice the resource requirement when the bit-width is doubled. Other examples for implementations with buses of full bit-width may be found in [13, 30–32].

The advantage of this approach is the nearly linear speed-up, which is generally obtainable with relatively small costs on this level. Drawbacks are the resource costs for parallel arithmetical units and communication path widths as well as a certain inflexibility concerning the possible bit-widths range. For smaller bit-widths than the maximum resources stay unused and larger bit-widths are usually infeasible. The solution from [3] somewhat avoids the latter problem by combining multiple ECC-cores into one RSA-core. However, it still exhibits unused resources for bit-lengths other than the bit-width of a single ECC-core and its multiples.

For implementations on FPGAs there is a further drawback stemming from their inflexibility concerning available memory. If several memory blocks are used in parallel to obtain a sufficient data-width, large parts of the resulting memory usually stay unused. Memory realized in logic cells of the FPGA is somewhat able to solve this problem, but at cost of cells, which are then not available for other tasks. This is explored in more detail in Section 3.1.

*Pipelining* Pipelining is in a way the complement to buses of full bit-width. It also can be applied to decrease the time complexity down to a linear behavior by increasing the throughput. Again, this comes at the cost of a linear increase in terms of resources. The bus-width is then usually chosen to be the word width. Using both buses of full bit-width and pipelining allows for a constant complexity. For example, the two-dimensional systolic array from [4] exhibits this property, as it uses buses of full bit-width and exploits as many pipeline stages as the operands have bits. Thus, after the pipeline is filled, in every cycle one final result is generated. But this design has two problems, as it suffers from above mentioned bit-width inflexibility and each pipeline stage has to be filled with data from a different modular multiplication. In [33] the same author proposes pipelining in an one-dimensional systolic array, which allows to reuse the stages in alternation for only two concurrent modular multiplication independent from each other.

Above pipelining solutions build upon the Montgomery multiplication and assign one pipeline stage to each step of the inner loop of the algorithm. As the first step is more complex than the following ones, this approach has the advantage that only the first stage has to be able to compute this first step. The remaining stages may be kept simpler. A drawback of the approach is a data feedback in the algorithm, which makes it nearly infeasible to use less stages than the number of words constituting the bit-width. Furthermore, for computations with less words than pipeline stages some stages stay unused.

This is addressed by the more flexible pipelined version of the Montgomery multiplication presented in [5]. It assigns one stage to each step of the outer loop and avoids the data feedback by keeping it inside a single stage. This allows a trade-off between resource usage and execution time, because the designer may choose an arbitrary amount of pipelining stages. However, now each stage has to be able to compute the more complex first step of the inner loop, thus forcing all stages to be as complex as the first.

Using latter approach, pipelining suffers less from the bit-width inflexibility than buses of full bit-width, as additional pipeline stages below a certain threshold, which depends on the bit-width, will increase throughput. The pipeline stages may be reused for the same multiplication operation, leading to an improvement even for just one instance of a modular multiplication.

*Karatsuba multiplication* The basic idea of the Karatsuba multiplication proposed in [34] relies on the fact that three word multiplications are sufficient to multiply two numbers with two words each, instead of the usual four operations by means of the School multiplication algorithm. This is shown in the following equation, where $w$ denotes the word-width and $x_h$ and $x_l$ refer to the high and the low word of $x$, respectively.

$$\begin{aligned} x \cdot y &= (x_h \cdot 2^w + x_l) \cdot (y_h \cdot 2^w + y_l) \\ &= x_h \cdot y_h \cdot 2^{2w} + ((x_h + x_l) \cdot (y_h + y_l) - x_h \cdot y_h \\ &\quad - x_l \cdot y_l) \cdot 2^w + x_l \cdot y_l \end{aligned}$$

Exploiting this fact in a recursive manner leads to a multiplication algorithm featuring a sub-quadratic complexity of $O(n^{\log_2 3}) \approx O(n^{1.585})$.

Since recursion is hard to implement in hardware, the Karatsuba multiplication is mostly applied for fixed bit-widths. [31, 32] use the Karatsuba idea for an improved multiplication of full bit-width and [8] exploits it for improved word multipliers combined with an enhanced multi-word multiplication algorithm. The algorithm from [7] is able to take arbitrary bit-lengths, but despite some improvements it still exhibits quadratic complexity. It depends on the constraints—mainly the cost difference between multiplication and addition and the computed bit-width—whether the advantage of the Karatsuba multiplication is large enough to compensate for the required increase in terms of number of additions and of control logic overhead.

*Residue Number Systems (RNS)* In this more unorthodox approach long integer numbers are represented relative to a base consisting of several smaller, relatively prime moduli. If $\{m_1, m_2, \ldots, m_n\}$ is such a set of relatively prime integers, every $x \in [0, 1, 2, \ldots, M-1]$ with $M = \prod_{i=1}^{n} m_i$ may be represented in an RNS by

$$x \Rightarrow \langle x \rangle_M = (\langle x \rangle_{m_1}, \langle x \rangle_{m_2}, \langle x \rangle_{m_3}, \ldots, \langle x \rangle_{m_n}),$$

where $\langle x \rangle_{m_i} = x \bmod m_i$. The Chinese Remainder Theorem ensures that there is an unique mapping between both representations. The advantage is that addition, subtraction, and multiplication operations may be performed on residue level, thus allowing the parallelization of the particular operation on up to $n$ arithmetical units as detailed in the following equations.

$$\begin{aligned}\langle x \pm y \rangle_M &= (\langle x_1 \pm y_1 \rangle_{m_1}, \langle x_2 \pm y_2 \rangle_{m_2},\\ &\quad \langle x_3 \pm y_3 \rangle_{m_3}, \ldots, \langle x_n \pm y_n \rangle_{m_n})\\ \langle x \cdot y \rangle_M &= (\langle x_1 \cdot y_1 \rangle_{m_1}, \langle x_2 \cdot y_2 \rangle_{m_2},\\ &\quad \langle x_3 \cdot y_3 \rangle_{m_3}, \ldots, \langle x_n \cdot y_n \rangle_{m_n})\end{aligned}$$

Note that the calculations in an RNS are implicitly executed modulo $M$.

Unfortunately, comparison and division/reduction can not be easily executed in RNS representation. However, for public key algorithms this constitutes no real disadvantage, as the reduction may be substituted by a multiplication with the inverse as done in the Montgomery multiplication. For further details see [9]. Associated hardware implementations may be found in [10, 11].

### 2.2. Parallelization on Elliptic Curve Group Level

This abstraction level considers elliptic curve addition and doubling, which are composed of modular operations. By assigning more parallel instances of the modular operations, the algorithms on this level may be accelerated, provided that the hardware architecture allows parallel execution. As these algorithms are independent of the bit-width, the problems of parallelization on the lowest level, namely bit-width inflexibility and resource utilization for different bit-widths, are not present here. However, because of data dependencies the usual algorithms on this level do not allow too many parallel instances and, thus, prohibit a truly linear speed-up with respect to the instance count.

For implementations of the point addition and doubling algorithms usually 2 or 3 parallel multipliers are suggested in literature [6, 12], as the data dependencies seem to limit further improvements seriously, see [13]. Furthermore, this abstraction level is well-suited for parallelizations in SIMD-type implementations as detailed in [12, 16]. But those results are not directly transferable to FPGAs, as their hardware architecture is limited to SIMD structures, whereas on FPGAs the designer can adjust the target architecture to her/his needs. For example, contrary to SIMD, a FPGA design may allow the parallel execution of modular additions and multiplications, see [2] or this work.

Remember that this abstraction level does not exist for RSA, because RSA uses the modular arithmetic directly as the multiplicative group for the modular exponentiation.

### 2.3. Parallelization on Cryptographic Main Operation Level

On this abstraction level, operations of the underlying group are employed to compute point multiplications (ECC) or modular exponentiations (RSA), respectively. Despite this difference, there are similarities between the exponentiation and the point multiplication. Therefore, both may be parallelized with similar methods.

For example, as suggested in [17], the Montgomery Ladder allows the utilization of two parallel instances of the group operations. With this approach only two parallel instances may be employed because of data dependencies. The approach from [29] permits the use of more parallel instances in combination with precomputation. For this purpose the scalar or the exponent, respectively, is split into several parts and every instance works on one of these parts starting with one of the precomputed values.

Although the benefits of parallelization are usually higher on lower levels, [19] utilizes parallel operations on this level too, as it offers additional improvements beyond those on the lower levels. However, parallelization on this level is mainly advocated for a better resistance against Side Channel Attacks (SCA). As the authors of [17] ([15] for SIMD) observe, parallel point addition and doubling helps making the execution time independent from the secret key. Reference [18] examines a method that splits doubling and addition into pipeline stages, thus allowing a partly overlapping execution of two EC operations. This leads to a better resistance against SCA too, as the authors of [18] point out.

Reference [20] contains a proposal to integrate the point addition and doubling into an atomic operation. Besides granting a better SCA resistance, this approach allows the use of more modular arithmetical units in parallel, because the atomic algorithm is more complex than usual point addition and doubling alone, see Section 4.5.

### *2.4. Parallelization on Cryptographic Scheme and Application Levels*

These two abstraction levels are mainly mentioned for the sake of completeness, as parallelization on these levels offers smaller benefits compared to the lower levels. Furthermore, most cryptographic schemes (an exception is, for example, the EC signature verification from IEEE P1363 [21]) contain only one point multiplication or modular exponentiation, respectively. Thus, they do not support parallelization directly. Consequently, we are not aware of any publication suggesting parallelization on these levels.

A possible scenario for parallelization on the application level would be a combined RSA/ECC coprocessor designed for different security levels and high throughput disregarding the latency of a single operation. Because of the huge difference between the bit-lengths of those protocols, parallelization on the modular arithmetic level is only possible to a low degree without accepting that allocated FPGA resources may stay unused for shorter bit-lengths. However, parallelization may still be performed on higher abstraction levels, e.g. by implementing multiple cryptographic scheme modules with relatively low resource usage and high latency. By working in parallel these allow the whole coprocessor to demonstrate a high throughput while being highly flexible in reference to possible bit-lengths and protocols.

## 3. Novel Memory Architecture

In this section the technical terms of the Xilinx architecture are used, because it seems to be one of the most common in literature. However, similar structures exist on devices of other manufacturers, too, see [24, 25].

FPGAs today implement boolean logic and registers as configurable logic blocks distributed over the whole device. In case of Xilinx FPGAs, these cells are further divided into so-called slices. In many architectures memory is available as plentiful Block RAM (BRAM) modules featuring two independent ports each. Special support for arithmetical operations is sometimes provided by units such as carry chains for additions and dedicated multipliers. All these resources are connected via a wire grid called routing resources, thus allowing to implement arbitrary circuitry.

### *3.1. Design Considerations*

The design goal of this work was aimed at an efficient implementation of ECC over $\mathbb{GF}(p)$ on an Xilinx Virtex II Pro FPGA. The design should be able to handle different security levels, i.e. different key lengths up to the maximum bit-width of 256. Its resource requirements should be relatively low, thus allowing the realization on smaller FPGA types or the integration of other functions on the FPGA as well (e.g., AES, random number generator, or something completely unrelated to cryptography). Consequently, the main objective was not a minimum execution time, but rather a small resource usage with a high utilization of allocated FPGA resources. This objective led to the following design considerations.

By committing to a FPGA as platform for the realization, the designer is restricted to the memory types offered on this platform. The Virtex II Pro offers two kinds of memory: Block RAM with a size of 18 Kbit each and SelectRAM+, which utilizes the Look-up Tables (LUT) and allows customizing memory-depth and -width. The ECC implementation needs approximately 1.7 KBytes of memory for all input, output, and intermediate values. This fits into just one BRAM. Employing SelectRAM+ it would take 2048 LUTs, because of constraints. As the objective was a low resource usage, it was decided to utilize BRAMs as memory. Note that in contrast to the BRAM the SelectRAM+ does not offer fully functional dual ports, i.e. while both ports may be read independently, only one may be written.

As explained in Section 2.1, buses of full bit-width are an efficient way to speed-up the computation. However, this approach hampers a flexible use of different key-lengths and consumes a larger amount of resources, mainly BRAMs, registers, and routing resources for the buses. In particular, allocated FPGA resources would stay unused for all computations with a bit-width less than the maximum, because the modular arithmetical units would have to be designed for the longest possible bit-width. Furthermore, for the effective use of buses with full bit-width, a memory with the same data-width is needed to read and write values in parallel. However, ECC (similar to RSA) requires mainly heavy computations, but has relatively low storage demands. The available BRAMs feature a maximum data bus-width of 32 bit, while offering a memory capacity of 2 KByte each. Therefore, to provide a memory with a data-width of 256 bit, at least 8 BRAMs are needed. Because the necessary data fits into only one BRAM, the majority of the memory will stay unused. Thus, because of the high resource requirements, the utilization of buses of full bit-width was rejected. Likewise, it was decided not to exploit pipelining, as the EC arithmetic should also be able to use, e.g., the algorithm from [7] for modular multiplication. Furthermore, it is not trivial to find the right amount of pipeline stages, if different security levels are to be processed by the same module.

Thus, parellelization was not implemented on the lowest level. But it may still be employed by other strategies, i.e. on higher abstraction levels. The next higher abstraction level is the elliptic curve group, see Section 2.2. On this level efficient parallelization can be done by using multiple instances of the modular multiplication, because this is the most important and longest operation. Therefore, as many instances as reasonable will be used and the architecture must allow those instances to run as continuously as possible. The remaining modular operations have substantially shorter execution times. Thus, a parallel implementation of these modular operations is not appropriate, as several of them may be executed consecutively during the runtime of one multiplication. This leads to an architecture, in which multiple multipliers are operating in parallel with respect to each other and to the remaining modular arithmetic as well.

A memory architecture supporting such an approach must be able to continuously feed data into all parallel arithmetical units to prevent stalling the execution of one unit, because data is not available at the right point in time. The conventional solution is to use one central memory and to equip every unit with input and output registers as depicted in Fig. 2, see [13, 14]. This allows the control logic to load new values into one arithmetical unit, while the others keep on working. Although this approach features a high simplicity, it requires input and output registers, which contain only data copied from memory, and additional cycles for copying this data. Therefore, it was not employed in the prototype implementation presented in this work.

For example, by using local input and output registers of 256 bit length each arithmetical unit needs additional 768 flipflops (2 input and 1 output registers), which requires 384 slices on the FPGA. This is not significant, when compared to the available resources, but the total amount increases considerably with the number of instantiated arithmetical units. Furthermore, assuming a word-width of 32 bit, it takes 16 cycles to copy both input values into the registers and further 8 cycles to copy the result from the output register during which the particular unit is not working. Fetching the values directly from memory when needed would obviously avoid such overhead.

### 3.2. Proposed Memory Architecture

It follows from the remarks in the last section that the proposed memory architecture has to provide a

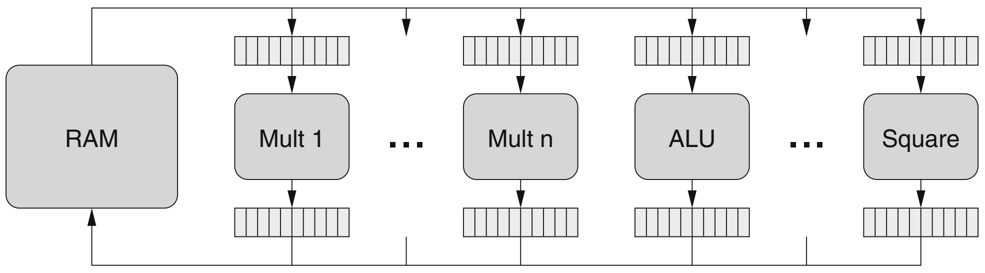

*Figure 2.* Conventional memory architecture.

continuous data flow and low resource usage. This is achieved by assigning each modular multiplier its own memory block connected via the first port. Thus, the multiplier does not share the memory access with parallel units and no registers are necessary to store values copied from or to the memory. The remaining modular arithmetic may access the data in the memory using the second port of each block. Note that this approach is not possible while employing SelectRAM+, because it offers only one writable port as described in Section 3.1.

The direct connection to memory keeps resource usage mainly in form of registers low and circumvents the necessity for copying, but exploits BRAMs, which are needed anyway to store both the start values and the end result. Furthermore, it allows the execution time to scale well with the number of modular multipliers. That is, by adding further modular multipliers, the execution time may be reduced as long as parallel execution is not inhibited by data dependencies within the algorithms.

A consequence of this direct connection between multiplier and memory block is that each multiplier can only access its own block. If a multiplier needs an intermediate result from another multiplier, then the remaining modular arithmetic has to copy this result into the correct memory block. Thus, the different memory blocks are held consistent by copying data between them. But the amount of copying can be kept reasonably low, as the storage area for temporary values does not need to be consistent at each point in time. For example, if consecutive multiplications are executed on the same multiplier, no copying of the intermediate results is necessary, as they are already stored in the proper memory block. Moreover, the need for copying does not affect scalability, as the remaining modular arithmetic is able to access multiple memory blocks in parallel. In the prototype implementation presented below no copying was needed outside the modular addition and subtraction operations. For these operations the common approach is to compute the normal and the reduced results in parallel and to store them in different memory locations. The final result $x$, which is one of these and lies in the range $0 \leq x < p$, is then copied and, thus, replaces the other result, which was stored

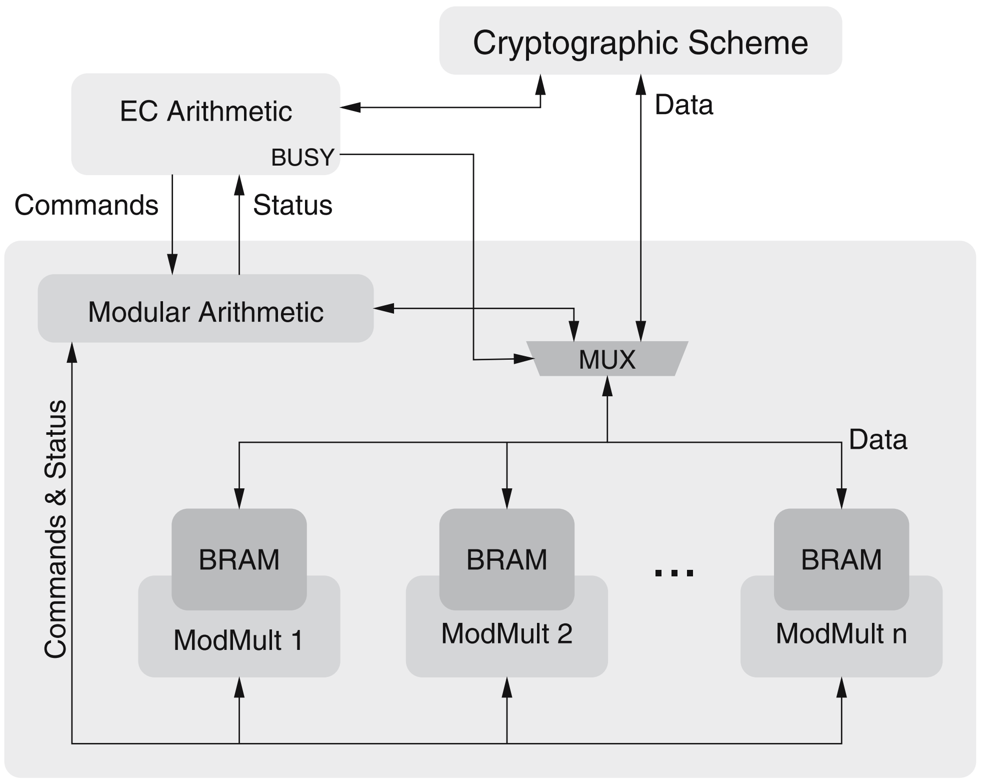

*Figure 3.* Distributed memory architecture for public key algorithms.

in a different memory block. Thus, the copying is executed as part of the addition, which may need it anyway, and in parallel to the multiplications. Hence, the scalability is not constrained.

A possible disadvantage is the increased amount of necessary BRAMs, which contain mainly redundant data. However, buses of full bit-width, for example, have an even higher memory requirement. Furthermore, the control logic for the EC addition and doubling algorithms becomes slightly more complicated, because it has to keep some parts of the memory blocks consistent and must be able to employ multiple modular multipliers in parallel. But as the control logic for the conventional architecture also has to copy data into the proper registers and employs several parallel arithmetic units, that increase is not too high. It may be noted that such additional work load does usually not cost execution time, as it can be done in parallel to the time-consuming multiplications.

The proposed architecture is depicted in Fig. 3, which also addresses the communication with higher abstraction levels. The EC Arithmetic, which encompasses the elliptic curve group and cryptographic main operation levels, uses the parallel modular arithmetical units to execute point addition and doubling and, in turn, the point multiplication. However, it does not need to access the integer values stored in the memory blocks directly. Rather, it modifies this data utilizing the modular operations. Thus, commands to the modular units are to be issued. In addition, the EC module needs to be able to read their status. To simplify this access, the channels for commands and status are unified: The EC arithmetic sends commands to different modular units—namely the different modular multipliers (ModMult in Fig. 3) and the remaining modular arithmetic—using the same communication path provided by the Modular Arithmetic. Furthermore, the EC arithmetic may read the execution status of all units from the modular arithmetic. This means, for example, that the EC arithmetic can start multiplications on each multiplier in parallel, then execute two additions in sequence on the modular arithmetic, and wait for the multipliers to finish prior to continuing with the next operation.

In contrast to the EC arithmetic, the Cryptographic Scheme has to access the integer values stored in the memory blocks directly. However, these accesses are only needed before the beginning and after the end of a point multiplication in order to write the parameter values and to read the results, respectively. Unfortunately, the BRAMs available on the Virtex II Pro lack a third port to easily facilitate this communication with the cryptographic scheme. But as these memory accesses are only executed while the EC arithmetic is not working anyway, the second port of the BRAMs may be reused employing the multiplexer MUX. Thus, while no point multiplication is executed, the BRAMs are accessible from the cryptographic scheme. During the time period a point multiplication is being executed, however, the modular arithmetic may access the BRAMs. Moreover, as the BRAMs are essentially copies of each other containing identical data—not counting temporary values—the access from the cryptographic scheme is simplified: From the cryptographic scheme all memory blocks are written in parallel with the same data and only one memory block may be read out.

Note that it is possible to implement BRAMs with a third and even fourth port by using time division multiplex access as described in [23]. But this may lead to a longer cycle period and to a higher resource usage. In fact, in another design we experienced that the cycle period is seriously degraded, if the size of the design approaches the size of the FPGA. Furthermore, disconnecting the cryptographic scheme from the memory during the runtime of a point multiplication actually increases robustness. However, depending on the application case, the usage of additional ports may be an approach well worth for additional examination.

## 4. Experimental Results

This section presents some demonstrators for the evaluation of the proposed architecture. It starts with a short introduction of the tool utilized to decide on the number of parallel multipliers and their schedule. Then the decision process using this tool is outlined. After motivating the decision on the amount of modular multipliers for the EC arithmetic specified in IEEE P1363, the results of the prototype implementation are presented and compared with those of other realizations. The section concludes with an application of the proposed architecture to another EC algorithm.

### *4.1. Design Space Exploration*

The proposed memory architecture allows the parallel utilization of several modular multipliers. However, data dependencies in the point addition and doubling algorithms restrict the number of usable units. Thus,

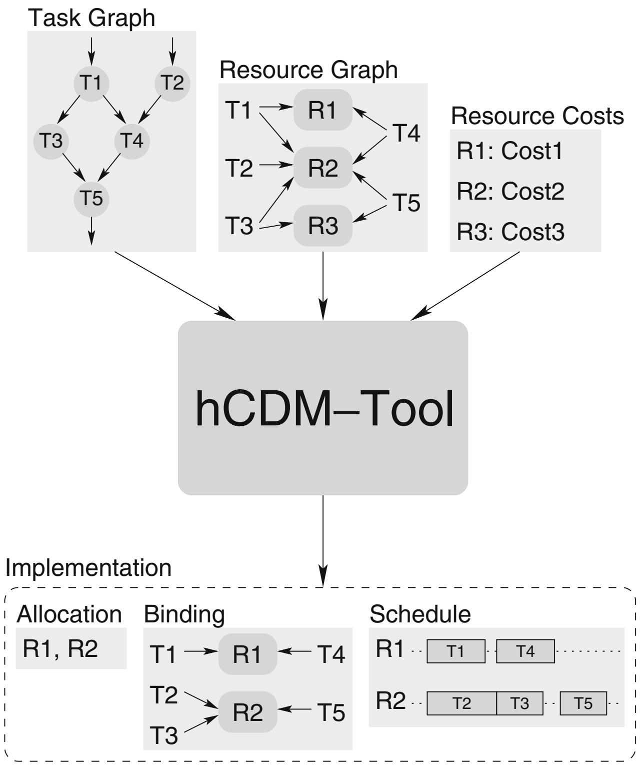

*Figure 4.* Outline of the work-flow of the hCDM-tool.

to avoid multipliers staying unused during longer periods of time, the designer has to find the appropriate amount and a feasible schedule. The complexity of this problem is NP-hard, i.e. it is not easily solvable in general.

Therefore, we suggest an automated approach, for which the high-level synthesis tool hCDM from [26–28] was used. Its function is the generation of optimized implementations of embedded systems by means of a genetic algorithm. Such an implementation consists of an allocation of resources out of the available set, of a binding of each task to an allocated resource, and of a schedule determining the chronological order of the tasks. Furthermore, the hCDM-tool provides optimization means according to different design criteria.

For the design of the prototype implementation in this work the hCDM-tool turned out to be very useful, because it generated feasible schedules for the point addition and doubling algorithms. Therefore, mainly the scheduling aspect was used.

The work-flow of the hCDM-tool is depicted in Fig. 4. The tool expects mainly information about tasks, about resources, about the data dependencies between the tasks, and about which task may be executed on which resource. In addition, further information may be used to optimize according to criteria like cost or maximized execution time for new tasks in updated task graphs. As this is beyond the scope of this work, this is only highlighted by the cost list for the different resources in Fig. 4, but not examined in detail. The tasks and their data dependencies are given in a task-graph similar to a data flow graph like the one depicted in Fig. 6. The information about the resources is given in a resource graph. This graph contains for every available

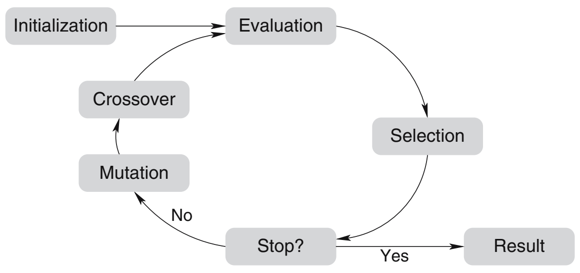

*Figure 5.* Outline of a genetic algorithm.

*Table 1.* Representation of an individual.

| | Allocation | Binding | Schedule |
|---|---|---|---|
| Purpose | Determine set of resources | Assign a resource to each task | Assign a priority to each task |
| Example | R1, R3, ... | (T1, R3), (T2, R1), ...,(Tn, R5) | (T1, 4), (T2, n), ...,(Tn, 1) |
| Mutate operation | Add/remove resources | Select new resource from allocation | Exchange priorities |

resource the information, which tasks may be executed on it accompanied with the estimated execution time of this task on this resource. From this information the hCDM-tool computes feasible allocations, bindings, and schedules with the help of a genetic algorithm.

A genetic algorithm is a heuristic search algorithm modeled after the theory of evolution using natural selection to find increasingly better solutions, which are also called individuals in this context. The basic process of a genetic algorithm is depicted in Fig. 5. At the beginning the Initialization function is used to randomly initialize a set of individuals, which is also called a generation. The individuals of this generation are then evaluated, i.e. assigned a score value according to criteria chosen by the designer by employing the Evaluation function. The Selection function then chooses individuals with high score values, which will constitute the foundation for the next generation. However, before the next generation is created, with a certain probability the chosen individuals are slightly modified by the Mutation function. The Crossover function, finally, produces the individuals for the next generation by combining the properties of its parents. With this new generation the process starts again with the evaluation step. The algorithm ends after a preset amount of generations or if a certain score level is reached for at least one individual.

An advantage of genetic algorithms is that they can search through a search space relatively fast, while being able to avoid local optima. Furthermore, they do not require sophisticated methods for finding good solutions, but mainly rely on a good representation for a solution and a realistic evaluation function.

Table 1 depicts the representation of an individual in hCDM. The allocation is a subset of the available resources, i.e. not all resources have to be used in the implementation. Out of the current allocation the binding assigns each task exactly one resource, on which this task has to be executed. The schedule, finally, describes the chronological order, in which the tasks are executed. For storage reasons this is done by assigning each task an unique priority.

A detailed description of all functions of the genetic algorithm used in the hCDM-tool is beyond the scope of this work, thus, as an example, only the mutation operation is explained here. The basic behavior of the mutation is shown in the last row of Table 1. The allocation is mutated by removing one randomly chosen resource. In the case that tasks exist, which can not be executed on the resulting set of resources,

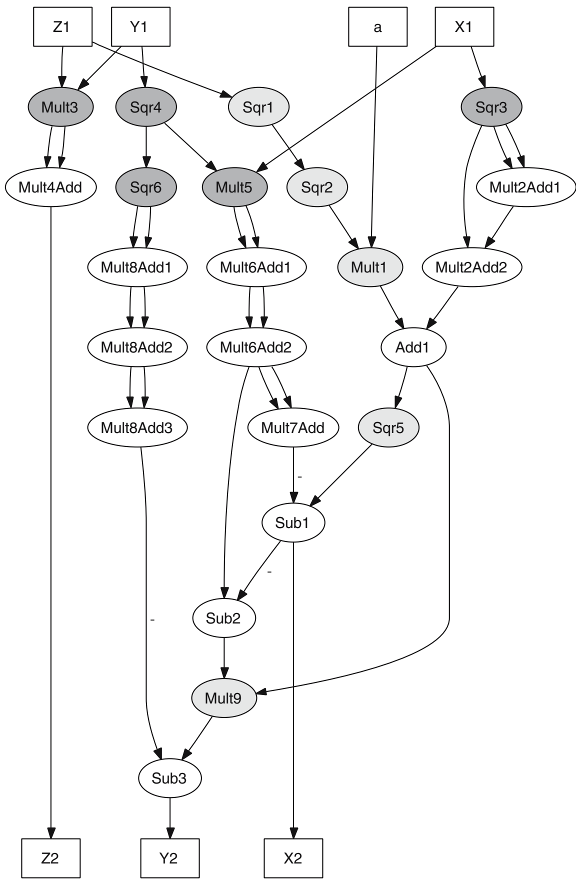

*Figure 6.* Data flow graph for IEEE EC point doubling.

the allocation is repaired by randomly adding resources, on which these tasks can be executed. The binding is mutated by assigning each task another resource from the allocation, if possible. The mutation of the schedule, finally, is done by exchanging the priorities of two randomly chosen tasks.

The use of the evaluation function makes genetic algorithms very flexible. By simply substituting this function, the search criteria may be changed completely. For example it is possible to optimize an implementation for execution time, costs, or other goals, even for combinations of them. In the hCDM-tool the score is given according to the Pareto-optimality of an individual. Thus, the score of an individual quantifies the amount of other solutions in the generation, which are dominated by this individual. Thereby, a solution dominates another, if it is equal or better in all criteria. The result of an optimization run of the hCDM-tool is a set of Pareto-optimal solutions, from which the designer then may chose the one fitting her/his needs best.

### 4.2. *Resource Allocation and Scheduling*

In case of the ECC prototype implementation the hCDM-tool was used to decide on the number of parallel multipliers to be utilized and to generate a feasible schedule. Possible resource types for the hCDM representation were the modular multipliers and the remaining modular arithmetic. The modular operations needed in the point addition and doubling, see Fig. 6, were denoted as tasks in the hCDM representation. For the decision on the number of multipliers the following approach was used. Thereby, the cycle counts for modular multiplication and addition/subtraction with a bit-width of 160, which are 598 and 41 cycles in our implementation, respectively, were used as the execution times for the tasks.

For all estimations one instance of the remaining modular arithmetic was employed, while a maximum amount of 5 multipliers was allowed, see below. The hCDM-tool was then utilized to produce a feasible schedule for different amounts of modular multipliers. The multiplier usage percentage of one solution is the part of the overall execution time, in which the multipliers are actually running, i.e. are not waiting for the remaining arithmetic.

After deciding on the number of multipliers utilized, the concerned schedule was refined further. To minimize necessary copying, multiplications working on the same values should be executed on the same multiplier, see Section 3.2. For this purpose the hCDM-tool was employed again, but at this time the possible bindings were restricted to enforce the assignment of multiplications belonging together to the same multiplier unit.

### 4.3. *IEEE P1363 Arithmetic Algorithm*

Figure 6 shows the data flow graph of the algorithm for point doubling in $\mathbb{GF}(p)$ produced from the IEEE standard in [21]. Operations like *Mult8Add1* denote multiplications with small constants, which are substituted by additions for efficiency. Parallel arrows denote the usage of the same value at both inputs. Only one type of modular multiplier is used for both multiplications and squarings.

The hCDM-tool was used to find optimal schedules as descried in Section 4.2. The results of an optimization run for the point doubling is shown as triangles in Fig. 7. It shows different solutions, which

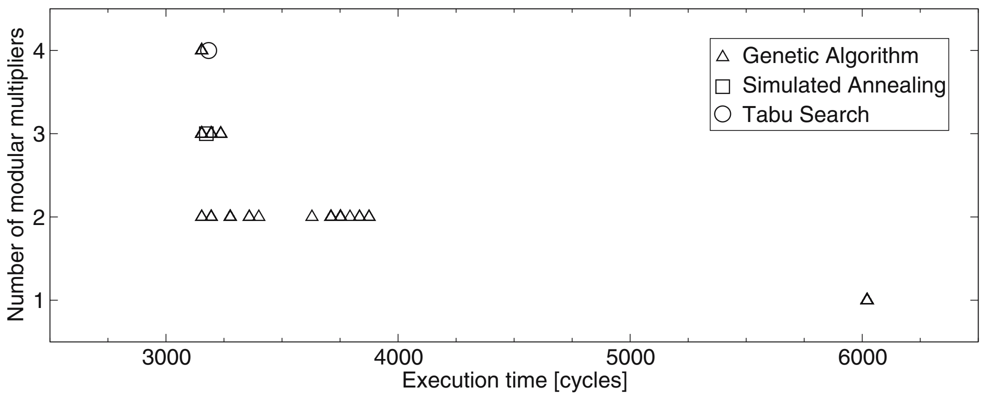

*Figure 7.* Results of different optimization algorithms for IEEE EC point doubling.

*Table 2.* Usage of multipliers for IEEE EC point addition.

| No. multipliers | Usage | No. consecutive multiplications |
|---|---|---|
| 2 | ca. 98% | 8 |
| 3 | ca. 82% | 6 |
| 4 | ca. 74% | 5 |

exhibit different execution times measured in cycles and differ in the modular multiplier count. This is also a good example for the property of the hCDM-tool to simultaneously generate multiple Pareto-optimal solutions.

The instantiation of two modular multipliers roughly halves the execution time compared to only one instance, while additional multipliers provide no further speed-up. This fact may also be seen from Fig. 6. The point addition needs a total of 10 modular multiplications and the longest path concerning modular multiplications is 5. Thus, a second modular multiplier may be exploited fully while the first one is working on its tasks on the longest path, but a third one would idle. The usage of 2 parallel multipliers resulted in an utilization of the multipliers of approximately 95%.

However, a question arises concerning the quality of the results produced by the outlined genetic algorithm when compared to other statistical approaches. This question was investigated for simulated annealing and for tabu search, see [38] and [37], respectively. These additional optimization approaches were implemented into the hCDM-tool, see [36]. Their results produced for the EC point doubling example are also depicted in Fig. 7. Obviously, both simulated annealing and tabu search find solutions featuring a nearly optimal solution time, although these solutions need three of even four modular multipliers, respectively. It is likely that the exploitation of other optimization weight values may have allowed both approaches to find a solution featuring the same execution time with only two multipliers, as well. However, the genetic algorithm simultaneously found almost optimal solutions for different amounts of modular multipliers, i.e. Pareto-optimal solutions. This demonstrates the quality of the representation of individuals, of the operations to generate new individuals, and of the evaluation function as detailed in Section 4.1.

For the more complex point addition algorithm in IEEE P1363 an in-depth examination was necessary. Thus, the hCDM-tool was used again to generate solutions with two, three, and four multipliers. The resulting usage percentages are shown in Table 2. The third column contains the number of multiplications, which have to be executed consecutively.

Table 2 also highlights the scalability of the proposed architecture. The execution time of the EC addition of 16 modular multiplication is halved by utilizing two multipliers. Three multipliers further decrease the execution time to a third of the smallest multiple of the number of multipliers being larger than the number of multiplications. An additional parallel multiplier reduces the amount of consecutive multiplications even more. However, the longest path of the algorithm consists of 5 modular multiplication. Thus, the number of consecutive multiplications can not be decreased further by utilizing five or more parallel multipliers. From these results it was decided to use two parallel multipliers only, as the usage percentage for three parallel multipliers was judged to be too low, especially as during point doubling the third one would idle.

In Section 3.2 it was proposed to execute multiplication using the same intermediate values on the same multiplier as a method for decreasing the amount of data copying. This was adopted in the prototype implementation as outlined in Fig. 6. The light and dark gray operation symbols refer to multiplications, where the color indicates on which of the two parallel multipliers the operation is being executed. The operation *Sqr1*, for example, stores its result in the memory block of multiplier *FFMultA*. Because *Sqr2* is executed on the same multiplier, the intermediate

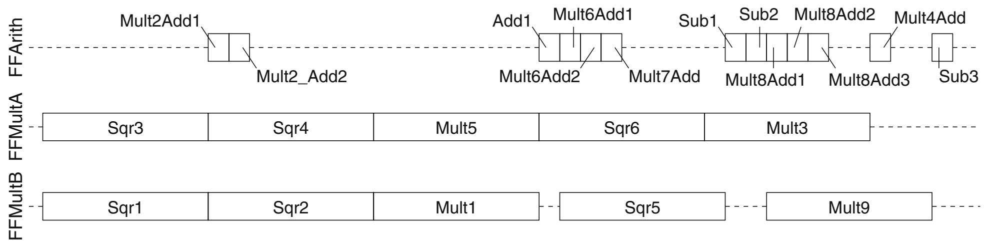

*Figure 8.* Schedule for IEEE EC point doubling executed on 2 multipliers.

*Table 3.* Experimental results and comparison with other implementations.

| Ref | Flip-Flops | LUTs | Slices | BRAMs | Cycle period | Point multiplication |
|---|---|---|---|---|---|---|
| this work | 1,128 | 3,015 | 1,806 | 3 | 9.898 ns | 12.716 ms (160 bit) |
| [2] | 6,959 | 11,227 | n/a | n/a | 10.952 ns | 14.414 ms (160 bit) |
| [30] | 5,735 | 11,416 | n/a | 35 | 25 ns | Estimated 3ms (192 bit) |
| [6] | n/a | n/a | 18314 | 24 | 100.1 ns | 114.71$\mu$s (191 bit $\mathbb{GF}(2^m)$) |

result does not need to be copied. Moreover, no copying was required in total except in the modular addition and subtraction algorithms, which always write their results into both BRAMs.

The final schedule for point doubling using two parallel multipliers named *FFMultA* and *FFMultB* generated by the hCDM-tool is shown in Fig. 8. It is not to scale, as the additions are depicted too long compared to the multiplications. The multipliers run nearly uninterrupted, while the remaining operations are executed concurrently.

## 4.4. Prototype Implementation

The prototype implementation was done on an XUPV2P Development System from Digilent, Inc. The central component on this board is a Xilinx Virtex II Pro FPGA (xc2vp30ff896-7), see [22]. The software tools used were *Synplify Pro 8.1* from Synplicity for the RT-level synthesis and the tools from the Xilinx Integrated Software Environment (ISE) 7.1 for Place&Route.

The point multiplication was implemented by means of a variant of the Lim/Lee exponentiation algorithm with 8 precomputed EC points, see [29]. This variant grants a better resistance against SCA, because it executes an operation even in case that all inspected bits of the scalar are 0. The modular multiplication was realized using the Montgomery multiplication algorithm taken from [35] without thorough optimization.

The resource usage and execution time of the prototype clocked with 100 MHz is shown in Table 3. Note that the execution time for one point multiplication on this prototype consists of the precomputation (6.434 ms) and the actual point multiplication (6.282 ms) based on the precomputed values. The precomputation, however, has to be executed only once for each set of curve parameters.

For comparison, the results of other published designs on top of Xilinx FPGAs can be found in Table 3. "n/a" indicates that a design result is not given in the respective publication. The other implementations were designed for the scalar multiplication of the given bit-length to be as fast as possible considering no other concurrent application on the FPGA. Therefore, all implementations exploit buses of full bit-width, thus they are not easily adaptable to different bit-widths. As this indicates a better parallelization than in our design, we expected our design to be significantly smaller, but also slower, i.e. not being downright superior, but exhibiting a better utilization of the allocated resources, in particular of the memory units.

Regarding [30] our expectation was fulfilled, as the increase in resources and the decrease in execution time is roughly proportional. The execution time was only estimated and is in reality most probably slightly higher, as a point multiplication does not consist of modular multiplications only. Furthermore, [30] seems to use a more efficient algorithm for the inversion in the finite field as it consumes just about 9% of the overall runtime, while our implementation needs about 15% of the overall computational effort. Especially the utilization of the BRAMs in [30] is poor, as the necessary data should fit well in two BRAMs (note that in [30] the BRAM size is 4 Kbit and its storage requirements sum up to approximately 0.9 KByte).

The design from [2] also needs much more resources than our design, because it features buses of full bit-

*Table 4.* Usage of multipliers for Algorithm 1.

| Ref. | No. multipliers | Multiplier usage | No. consecutive multiplications | No. consecutive additions |
|---|---|---|---|---|
| [20] | 2 | ca. 90% | 10 | 8 |
| this work | 2 | ca. 94% | 10 | 1 |
| | 3 | ca. 90% | 7 | 1 |
| | 4 | ca. 89% | 5 | 5 |
| | 5 | ca. 75% | 5 | 1 |

**Algorithm 1** Atomic "*EC double and add*" algorithm from (20)

```
Input: coordinates $X_P, Z_P, X_Q, Z_Q$; curve parameters $a, b$; x-coordinate of base point $X_D$
Output: coordinates $X_{P'}, Z_{P'}, X_{Q'}, Z_{Q'}$
 1:  $R0 \leftarrow X_P, R1 \leftarrow Z_P, R2 \leftarrow X_Q, R3 \leftarrow Z_Q$
 2:  $R6 \leftarrow R2 \cdot R1$        (1)   |  $R7 \leftarrow R3 \cdot R0$       (2)
 3:  $R4 \leftarrow R7 + R6$        (3)   |  $R5 \leftarrow R7 - R6$       (4)
 4:  $R5 \leftarrow R5 \cdot R5$        (5)   |  $R7 \leftarrow R1 \cdot R3$       (6)
 5:  $R1 \leftarrow a \cdot R7$         (7)   |  $R6 \leftarrow R7 \cdot R7$       (8)
 6:  $R0 \leftarrow R0 \cdot R2$        (9)   |  $R6 \leftarrow b \cdot R6$        (10)
 7:  $R0 \leftarrow R0 + R1$        (11)  |  $R6 \leftarrow R6 + R6$       (12)
 8:  $R0 \leftarrow R0 \cdot R4$        (13)  |  $R1 \leftarrow x_D \cdot R5$      (14)
 9:  $R4 \leftarrow R0 + R6$        (15)  |  —
10:  $R4 \leftarrow R4 + R4$        (16)  |  $R6 \leftarrow R2 + R2$       (17)
11:  $R4 \leftarrow R4 - R1$        (18)  |  $R7 \leftarrow R3 + R3$       (19)
12:  $R0 \leftarrow R6 \cdot R7$        (20)  |  $R1 \leftarrow R3 \cdot R3$       (21)
13:  $R2 \leftarrow R2 \cdot R2$        (22)  |  $R3 \leftarrow a \cdot R1$        (23)
14:  $R6 \leftarrow R2 - R3$        (24)  |  $R7 \leftarrow R2 + R3$       (25)
15:  $R1 \leftarrow R1 + R1$        (26)  |  —
16:  $R2 \leftarrow b \cdot R1$         (27)  |  $R7 \leftarrow R7 \cdot R0$       (28)
17:  $R1 \leftarrow R2 \cdot R1$        (29)  |  $R0 \leftarrow R0 \cdot R2$       (30)
18:  $R6 \leftarrow R6 \cdot R6$        (31)  |  —
19:  $R6 \leftarrow R6 - R0$        (32)  |  $R7 \leftarrow R7 + R1$       (33)
20:  $X_{P'} \leftarrow R4, Z_{P'} \leftarrow R5, X_{Q'} \leftarrow R6, Z_{Q'} \leftarrow R7$
```

width connecting operations and registers. It does not use BRAMs, thus eliminating extra cycles for copying. Surprisingly, this design is slower than our prototype. Although we are not entirely sure, we assume that this is because our modular multiplication was implemented with a simple Montgomery multiplication using the dedicated multipliers of the FPGA, while [2] implemented the modular multiplication by means of a systolic array without using the dedicated multipliers. Thus, the complexity in our design is quadratic relative to the number of words, while the complexity of [2] is linear to the number of bits. Therefore, it should be faster than our design for higher bit-widths, but at the cost of a further increase in resource usage. Note that the design from [2] does not only implement parallelization on the lowest abstraction level, but also on the elliptic curve group level: Modular multiplication and addition can be executed in parallel.

Although the design from [6] is based on $\mathbb{GF}(2^m)$, it is included for comparison, because it uses parallelization on the elliptic curve group level in a way similar to our design. It features two logical memory banks each built from several BRAMs directly connected to two modular arithmetical units, thus no additional registers or additional cycles for copying are required. Each of the two ports of the two logical memory banks is directly connected to one of the arithmetical units consisting of a multiplier and an adder connected in series, thus allowing the units to fetch data from both banks. But as one BRAM has only two ports, this solution does not scale well for additional multipliers. Furthermore, it is not easily transferable to $\mathbb{GF}(p)$, because the modular operations in $\mathbb{GF}(p)$ are gener- ally more complicated. Separating the multiplier and the adder seems to be a good design decision for $\mathbb{GF}(p)$. Similar to [30], it

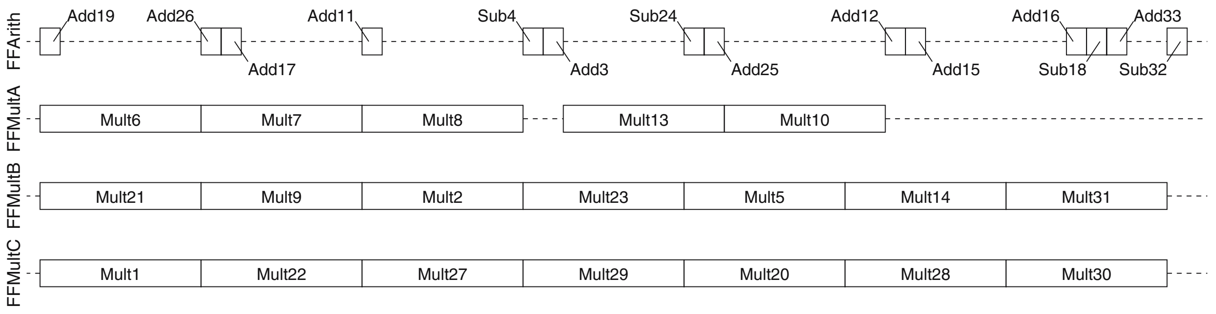

*Figure 9.* Schedule for Algorithm 1 executed on 3 parallel multipliers.

exhibits a poor utilization of the allocated memory blocks.

### *4.5. Application to Another EC Arithmetic Algorithm*

For further examination of the applicability of the proposed memory architecture to other algorithms, the method for atomic point doubling and addition detailed in [20] was investigated. Algorithm 1 outlines this method in the proposed form for two parallel modular arithmetical units. Addition and multiplication are never executed in parallel, because the underlying hardware architecture is SIMD-like and does not allow it.

As this algorithm contains more modular operations than the EC algorithms from IEEE P1363, we assumed that it is suited for a reasonable use of more parallel units, i.e. modular multipliers. Based on its data flow graph, schedules for different numbers of multipliers were generated (again with the hCDM-tool). Table 4 shows the resulting multiplier usage percentages. The last column contains the number of additions, which increase the overall runtime, i.e. which are not executed in parallel to a multiplication. Because cycle counts were not given in [20], the cycle counts from this work were used to calculate the multiplier usage value in the first line of Table 4.

As expected, more multipliers may be reasonably applied compared to the IEEE point addition and doubling. For three multiplier instances the utilization is still at 90%. The resulting point multiplication would thus be much faster: An implementation of the atomic operation as given in [20] utilizing three multipliers takes 7 consecutive multiplications. This is shorter than the runtime of one point addition from our IEEE implementation alone, not yet accounting for the point doubling needed in our design additionally. Note that the improvement compared to the IEEE operations results partly from the shorter algorithm in [20]: It needs only 19 modular multiplications for the atomic operations, while point addition and doubling as outlined in [21] need a total of 26 multiplications.

A schedule using three parallel multiplier instances produced by the hCDM-tool is shown in Fig. 9. Except in modular addition and subtraction no copying between the memories is necessary, because multiplications operating on the same values may be executed on the same multiplier instance. Like in the schedule for EC doubling in Fig. 8, the multipliers run nearly continuously and all remaining additions and subtractions are executed in parallel.

## 5. Conclusion

Starting from a systematic review of parallelization methods on different abstraction levels of public key cryptography, we presented a new memory architecture for ECC implementations on FPGAs using multiple modular multipliers operating in parallel. The proposed architecture allows the execution time to scale well with the number of multipliers and circumvents the usage of registers by accessing the memory directly. Although it complicates the control logic for the EC arithmetic slightly, it limits the overall execution time nearly to the mere runtime of the multipliers by executing the remaining modular operations in parallel. We demonstrated the suitability of this architecture by presenting an IEEE EC implementation, which features a comparable efficiency to previous work, while exhibiting a better utilization of the allocated FPGA resources, in particular of the memory. The applicability of the proposed architecture to other algorithms was demonstrated by adopting it for a different EC algorithm taken from literature.

**Acknowledgements**

The authors thank the anonymous reviewers for their comments how to further improve the article. Furthermore, the authors thank Abdulhadi Shoufan for many helpful discussions.

## References

1. Siddika Berna Örs, Lejla Batina, Bart Preneel, and Joos Vandewalle, “Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array,” *International Parallel and Distributed Processing Symposium—IPDPS*, 2003, pp. 184–191.
2. Siddika Berna Örs, Lejla Batina, Bart Preneel, and Joos Vandewalle, “Hardware Implementation of an Elliptic Curve Processor over $\mathbb{GF}(p)$, IEEE International,” *Conference on Application-Specific Systems, Architectures, and Processors—ASAP*, 2003, pp. 24–26.
3. Francis Crowe, Alan Daly, and William P. Marnane, “A Scalable Dual Mode Arithmetic Unit for Public Key Cryptosystems,” *International Conference on Information Technology: Coding and Computing—ITCC*, Volume 1, 2005, pp. 568–573.
4. Colin D. Walter, “Systolic Modular Multiplication,” *IEEE Trans. Comput.*, vol. 42, no. 3, 1993, pp. 376–378.

5. Alexandre F. Tenca and Ç.K. Koç, “A Scalable Architecture for Modular Multiplication Based on Montgomery’s Algorithm,” *IEEE Trans. Comput.*, vol. 52, no. 9, 2003, pp. 1215–1221.
6. Nazar A. Saqib, Francisco Rodríguez-Henríquez, and Arturo Díaz-Pérez, “A Parallel Architecture for Computing Scalar Multiplication on Hessian Elliptic Curves,” *International Conference on Information Technology: Coding and Computing—ITCC*, Volume 2, 2003, pp. 493–497.
7. Rainer Blümel, Ralf Laue, and Sorin A. Huss, “A Highly Efficient Modular Multiplication Algorithm for Finite Field Arithmetic in $\mathbb{GF}(p)$,” ECRYPT Workshop: CRyptographic Advances in Secure Hardware–CRASH, 2005.
8. Markus Ernst, Michael Jung, Felix Madlener, Sorin A. Huss, and Rainer Blümel, “A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over $\mathbb{GF}(2^m)$,” Workshop on Cryptographic Hardware and Embedded Systems—CHES, ser. *Lect. Notes Comput. Sci.*, vol. 2523, 2002, pp. 381–399.
9. Jean-Claude Bajard and Laurent Imbert, “A Full RNS Implementation of RSA,” *IEEE Trans. Comput.*, vol. 53, no. 6, 2004, pp. 769–774.
10. Mathieu Ciet, Michael Neve, Eric Peeters, and Jean-Jacques Quisquater, “Parallel FPGA Implementation of RSA with Residue Number Systems—Can side-channel threats be avoided?,” *IEEE Midwest International Symposium on Circuits and Systems—MWSCAS*, vol. 2, 2003, pp. 806–810.
11. Dimitris Schinianakis, A.P. Fournaris, Athanasios P. Kakarountas, T. Stouraitis, “An RNS Architecture of an $F_p$ Elliptic Curve Point Multiplier,” *IEEE Int. Symp. Circuits Syst.—ISCAS*, 2006, pp. 3369–3373.
12. Kazumaro Aoki, Fumitaka Hoshino, Tetsutaro Kobayashi, and Hiroaki Oguro, “Elliptic Curve Arithmetic Using SIMD,” International Conference on Information Security—ISC, ser. *Lect. Notes Comput. Sci.*, vol. 2200, 2001, pp. 235–247.
13. Marcus Bednara, M. Daldrup, Joachim von zur Gathen, Jamshid Shokrollahi, and Jürgen Teich, “Reconfigurable Implementation of Elliptic Curve Crypto Algorithms,” *International Parallel and Distributed Processing Symposium—IPDPS*, 2002, pp. 157–164.
14. Guido Bertoni, Luca Breveglieri, Thomas J. Wollinger, and Christof Paar, “Finding Optimum Parallel Coprocessor Design for Genus 2 Hyperelliptic Curve Cryptosystems,” *International Conference on Information Technology: Coding and Computing—ITCC*, 2004, pp. 538–544.
15. Tetsuya Izu and Tsuyoshi Takagi, “A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks, ” International Workshop on Practice and Theory in Public Key Cryptosystems: Public Key Cryptography, ser. *Lect. Notes Comput. Sci.* vol. 2274, 2002, pp. 280–296.
16. Tetsuya Izu and Tsuyoshi Takagi, “Fast Elliptic Curve Multiplications with SIMD Operations,” International Conference on Information and Communications Security—ICICS, ser. *Lect. Notes Comput. Sci.*, vol. 2513, 2002, pp. 217–230.
17. Marc Joye and Sung-Ming Yen, “The Montgomery Powering Ladder,” Workshop on Cryptographic Hardware and Embedded Systems—CHES, ser. Lect. *Notes Comput. Sci.*, vol. 2523, 2002, pp. 291–302.
18. Pradeep Kumar Mishra, “Pipelined Computation of Scalar Multiplication in Elliptic Curve Cryptosystems,” Workshop on Cryptographic Hardware and Embedded Systems—CHES, ser. *Lect. Notes Comput. Sci.*, vol. 3156, 2004, pp. 328–342.
19. Nadia Nedjah and Luiza de Macedo Mourelle, “Reconfigurable Hardware Implementation of Montgomery Modular Multiplication and Parallel Binary Exponentiation,” *Euromicro Symposium on Digital Systems Design—DSD*, 2002, pp. 226–235.
20. Wieland Fischer, Christophe Giraud, and Erik Woodward Knudsen, “Parallel Scalar Multiplication on General Elliptic Curves Over $F_p$ hedged against Non-Differential Side-Channel Attacks,” Cryptology ePrint Archive, Report 2002/007, IACR, 2002.
21. IEEE 1363, “Standard Specifications for Public-Key Cryptography—Annex A,” http://grouper.ieee.org/groups/1363/, 2000.
22. XILINX, “Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheets,” http://www.xilinx.com/products/, 2005.
23. Nick Sawyer and Marc Defossez, “Quad-Port Memories in Virtex Devices, XILINX,” http://direct.xilinx.com/bvdocs/appnotes/xapp228.pdf, 2002.
24. Atmel, “AT40K05/10/20/40AL Summary,” http://www.atmel.com/dyn/resources/prod_documents/2818s.pdf, 2004.
25. QuickLogic, “Eclipse Family Data Sheet (Rev. D),” http://www.quicklogic.com/images/Eclipse_Family_DS.pdf, 2005.
26. Arshad Jhumka, Stephan Klaus, and Sorin A. Huss, “ A Dependability-Driven System-Level Design Approach for Embedded Systems, Design, Automation and Test in Europe—DATE,” 2005, pp. 372–377.
27. Stephan Klaus, “System-Level Design Methodology for Embedded Systems (in German),” Technische Universtität Darmstadt, Computer Science Department, PhD Thesis, 2005.
28. Stephan Klaus and Sorin A. Huss, “Konzepte zur Beherrschung der Entwurfskomplexität eingebetteter Systeme (Concepts for the Control of the Complexity of Embedded System Design),” it—Information Technology, Methoden und innovative Anwendungen der Informatik und Informationstechnik, vol. 46, no. 2, 2004, pp. 59–66.
29. Chae Hoon Lim and Pil Joong Lee, “More Flexible Exponentiation with Precomputation,” Advances in Cryptography—CRYPTO, ser. *Lect. Notes Comput. Sci.*, vol. 839, 1994, pp. 95–107.
30. Gerardo Orlando and Christof Paar, “A Scalable $\mathbb{GF}(p)$` Elliptic Curve Processor Architecture for Programmable Hardware,” Workshop on Cryptographic Hardware and Embedded Systems—CHES, ser. *Lect. Notes Comput. Sci.*, vol. 2162, 2001, pp. 348–363.
31. Francisco Rodríguez-Henríquez and Çetin Kaya Koç, “On Fully Parallel Karatsuba Multipliers for $\mathbb{GF}(2^m)$,” International Conference on Computer Science and Technology—CST, 2003, pp. 405–410.
32. Nadia Nedjah and Luiza de Macedo Mourelle, “Fast Less Recursive Hardware for Large Number Multiplication Using Karatsuba-Ofman’s Algorithm, ”*Comput. Inf. Sci.*—ISCIS, 2003, pp. 43–50.
33. Colin D. Walter, “Improved Linear Systolic Array for Fast Modular Exponentiation,” *IEE Proc. Comput. Digit. Tech.*, vol. 147, no. 5, 2000, pp. 323–328.
34. Anatolii Karatsuba and Yu Ofman, “Multiplication of Multidigit Numbers on Automata,” *Sov. Phys.*—Doklady (Engl. transl.), vol. 7, no. 7, 1963, pp. 595–596.
35. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, “Handbook of Applied Cryptography”, CRC Press, 1997.
36. Mikaël Després, “Comparison of Optimization Algorithms Aimed to Design Space Exploration of Embedded Systems

(in German)," Technische Universität Darmstadt, Computer Science Department, Diploma Thesis, 2006.

37. Fred Glover and Fred Laguna, "Tabu Search," Kluwer Academic Publishers, Norwell, MA, USA, 1997.
38. Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi, "Optimization by Simmulated Annealing," *Science*, vol. 220, no. 4598, 1983, pp. 671–680.

**Ralf Laue** studied computer science at Technische Universität Darmstadt, Germany. In 2003, he received the Dipl.-Inform. degree. Since then he worked within the research program SicAri as research assistant at the Integrated Circuits and Systems Laboratory at the department of computer science at Technische Universität Darmstadt. He authored or co-authored 7 reviewed conference papers. His current research interest is dedicated hardware for cryptography focussing on parallelization in public key cryptography systems. He is member of the German Computer Science Association (GI).

**Sorin A. Huss** studied electrical engineering at Technische Universität München, Munich, Germany. In 1976, he received the Dipl.-Ing. degree and in 1982 the Dr.-Ing. degree. He worked from 1982 until 1990 with AEG Aktiengesellschaft in Ulm, Germany, as the CAD/CAE manager of the AEG Integrated Circuits Design Center. Since 1990, Dr. Huss is a full professor in the Computer Science Dept. of Technische Universität Darmstadt, Germany, and, in addition, a faculty member of the Electrical Engineering Dept. of the same university. He authored or co-authored two books, several book chapters, and more than 140 reviewed journal and conference papers; he received several Best Paper Awards. His current research interests are in the areas of embedded systems engineering and of reconfigurable HW/SW architectures for IT security applications. Prof. Huss is a member of ACM, of IEEE, of the German Computer Science Association (GI), and of the German Information Technology Society (ITG). Dr. Huss was the General Chair of the FDL'06 Conference and served as a member of many conference program committees and editorial boards.