# Power Aware Reconfigurable Multiprocessor for Elliptic Curve Cryptography

Madhura Purnaprajna, Christoph Puttmann, Mario Porrmann
Heinz Nixdorf Institute, Systems and Circuit Technology
University of Paderborn, Germany
{madhurap, puttmann, porrmann}@hni.upb.de

## Abstract

*Reconfigurable architectures are being increasingly used for their flexibility and extensive parallelism to achieve accelerations for computationally intensive applications. Although these architectures provide easy adaptability, it is so with an overhead in terms of area, power and timing, as compared to non-reconfigurable ASICs. Here, we propose a low overhead reconfigurable multiprocessor, which provides both parallelism and flexibility. The architecture has been evaluated for its energy efficiency for a computational intensive algorithm used in elliptic curve cryptography (ECC).*

*Typically, algorithms in ECC exhibit task-level parallelism and demand large amount of computational resources for custom implementations to achieve a significant speedup. A finite field multiplication in $GF(2^{233})$ was chosen as a sample application to evaluate the performance on the QuadroCore reconfigurable multiprocessor architecture. A three-fold performance improvement as compared to a single processor implementation was observed. Further, via reconfiguration to suit the application, power savings of about 24% were noted in UMC's 90nm standard cell technology.*

## 1 Introduction

Quantifying the characteristics of a processor implies comparing its area, operating frequency and power with other existing processor architectures. For reconfigurable processors, the performance is mainly judged by measuring the overhead incurred on account of reconfiguration, i.e. the additional area, timing and the impact on power. For runtime reconfigurable processors, the time required to reconfigure is also a performance overhead. Further, the performance of an application when mapped on to this architecture influences application specific characteristics such as execution time, total power consumption and energy consumption. Typically, reconfigurability is introduced to reduce the time to market and to introduce architectural flexibility. Primarily, runtime reconfigurability is employed for quick design modifications and also to reuse the reconfigurable area. Here, runtime reconfiguration is used as a method to alter power or energy characteristics. We propose a scheme of reconfiguration that allows our QuadroCore reconfigurable multiprocessor to switch between a fixed set of reconfigurable operating modes. These modes allow altering the architecture to suit the method of synchronization, communication and type of parallelism, as required by the different parts of a single application. The advantages of these reconfigurable operating modes include power and energy savings. Also, the proposed method of reconfiguration is quick and hence time efficient.

Many applications require secure communication in order to exchange sensitive information. To establish a secure communication channel, typically public-key cryptography is used. In this work, we focus on public-key cryptography based on elliptic curves over binary extension fields $GF(2^m)$. Compared to the frequently used RSA algorithm, elliptic curve cryptography (ECC) achieves the same security level as RSA with significantly shorter key sizes. However, algorithms for ECC are computational intensive. Especially multiplication in finite fields represents one of the most critical operations [1]. As a case study, we explore an algorithm based on ECC on our QuadroCore architecture. Therefore, we map a parallelized finite field multiplication to the processors of our QuadroCore. While applying different reconfiguration modes, the performance of the distributed multiplication is analyzed in terms of execution time and power consumption. Here, the execution time also includes the reconfiguration time involved.

The rest of the paper is organized as follows: Section 2 provides a brief description of existing implementation of ECC-based algorithms. Next, a comparison between existing reconfigurable processor architectures and our QuadroCore architecture is provided. Section 3 details the architecture of our QuadroCore and the modifications required to introduce runtime reconfiguration. The implementation scenario in terms of algorithmic details and modifications

978-3-9810801-3-1/DATE08 © 2008 EDAA

to the algorithm to suit the QuadroCore are mentioned in Section 4. Section 5 provides the detailed results in terms of timing and power reports. Finally, the conclusions are discussed in Section 6.

## 2 Existing Multiprocessors

A lot of research considers the efficient implementation of ECC algorithms on processor-based systems. Predominantly, the execution time is optimized by applying specialized algorithms on general purpose uni-processors [5]. Hardware accelerators have also been used in ECC to reduce the total execution time. Here, either dedicated coprocessors have been used or instruction set extensions have been added to the base processor [12]. However, in all these implementations the power consumption is usually not analyzed. Here, we propose to introduce reconfigurability within the QuadroCore multiprocessor organization to allow co-operative operation of the processing elements, which results in energy savings.

In most multiprocessors, suggestions for application partitioning and load distribution depend entirely on the programmer. This process of partitioning an application to multiple processors involves inter-processor communication. Further, establishing communication between processors necessitates use of predetermined methods of synchronization. The methods of synchronization and communication depend largely on the granularity of processing, i.e., instruction, data or task-level parallelism. Multiprocessor architectures are usually suited for a fixed granularity. For example, architectures such as Ambric [4], Tilera [11] and multicore network processors are well suited for data parallel applications. Hence, the synchronization and communication mechanisms are asynchronous streams of instruction and data. Consequently, a multiprocessor environment is most often a collection of uni-processors operating independently. Unlike these architectures, the QuadroCore can be adapted according to the applications' parallelism. This is achieved by reconfiguring the processors at runtime to suit instruction level, data level or task level parallelism, as per the application mapped. Another architecture that configures to varying granularities and parallelism as per work loads, called TRIPS, is presented in [10]. The architecture is composed of large coarse-grained components, which are partitioned and modular in the processor and memory subsystems. Point to point communication channels allows exposure to software for optimization. Unlike TRIPS, our QuadroCore is based on introducing reconfigurability to legacy processors when used for co-operative multiprocessing. Flexibility is achieved by adding a reconfigurable interconnect to the multiprocessor architecture. In addition, this limited modification allows retaining the same instruction set architecture. Further, co-operative multiprocessing is achieved by switching between a fixed set of reconfigurable operating modes, as suggested by the application.

## 3 QuadroCore Architecture

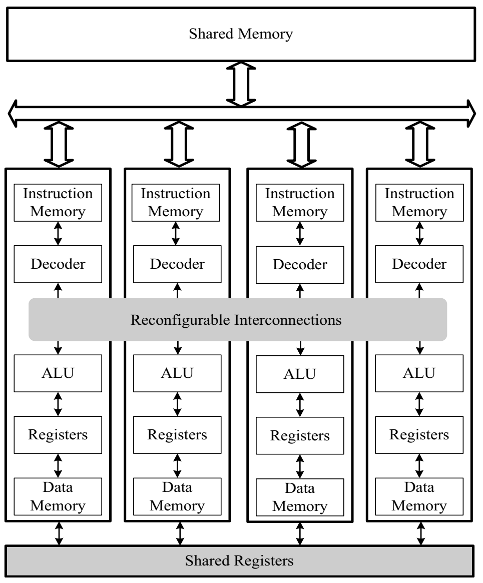

**Figure 1. QuadroCore Architecture**

A high-level representation of the QuadroCore reconfigurable multiprocessor is shown in Figure 1. It is composed of 32-bit RISC-based processors, described in [3], typically targeted for network processing applications. A combination of four processors forms a cluster. The default mode of operation is the MIMD mode, where each processor operates independently. Further, this hierarchy of processors in a cluster can be extended to multiple clusters interconnected by a network on chip.

### 3.1 Reconfiguration Mechanism

Reconfiguration in this context refers to runtime architectural alterations between a fixed set of modes to introduce enhanced performance during the execution of an application. This scheme of reconfiguration is scheduled during compilation, where the compiler inserts special reconfiguration instructions to mark the boundaries between the predetermined modes. The details of the compiler initiated methods can be found in [6]. When encountered with this special instruction, the reconfigurable interconnect is switched to suit the mode. The main advantages of this approach are the automatic management of reconfiguration via the compiler, fast single cycle reconfiguration and saving in configuration space achieved by embedding the configuration information in the instruction stream. Table 1 lists the

**Table 1. Reconfigurable Operating Modes**

| Operating Mode | Application Characteristics |
|---|---|
| Asynchronous MIMD | Coarse grained, Task level parallelism |
| Synchronous MIMD | Fine grained, Instruction level parallelism |
| SIMD | Low Power, Data level parallelism |
| Fast Memory Access | Large amount of data exchange, Data level parallelism, |
| Communication via Shared Register File | Few, frequent register exchange, Fine grained |

possible reconfigurable operating modes and the variations between them.

As seen from the table, based on these operating modes, QuadroCore supports parallelism at variable granularities. This feature demands a procedure to vary the method of communication and synchronization, as and when required by the application. These variations in granularity could be within different blocks of an application or between multiple applications. The following section describes the mechanism of these operating modes, viz., synchronization, communication, Multiple Instruction Multiple Data (MIMD) and the Single Instruction Multiple Data (SIMD) modes.

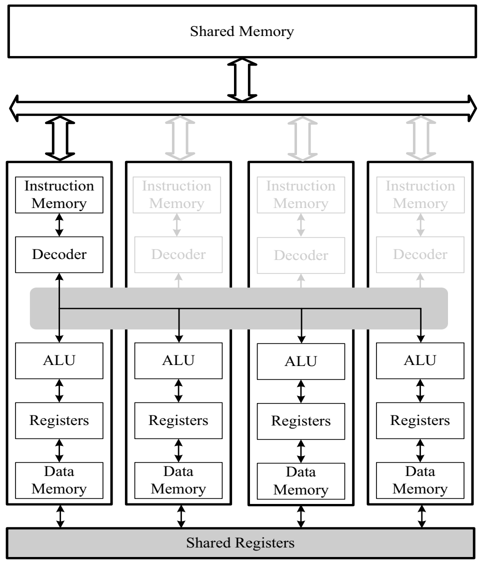

**Figure 2. SIMD Mode**

### 3.1.1 MIMD

In this mode of operation, all the processors operate on independent instruction and data streams, hence termed multiple instruction multiple data mode. This mode is suited for applications with coarse grained (like task level parallelism), with independent instruction and data streams. Depending on the frequency of data exchange, a synchronization scheme can be chosen to suit the application.

### 3.1.2 SIMD

As the name suggests, all the processors operate on a single instruction stream. Of the four independent instruction streams, the choice of the 'master' instruction stream is made at run-time and the corresponding 'master' processor performs the instruction fetch and decode for the remaining processors. Thus, in this mode, the instruction fetch and decoding stages for the other instruction streams is unused and hence is powered off, making a direct impact for power savings. Also, the required instruction memory space for these processing elements is reduced, since a single instruction stream is broadcast to all the elements.

### 3.1.3 Synchronization

A variable granularity of parallelism demands suitable synchronization schemes, such that the clock-cycle overhead of synchronization is minimal. Here, two modes of synchronization are supported. For infrequent or coarse-grained parallelism, synchronization between processors is achieved via single-cycle barrier synchronization. This mode is termed as asynchronous MIMD, since the processors operated independently until encountered by barriers. For frequent or fine-grained synchronization, the processors can be configured during runtime to operate in lock-step fashion. In this mode, all the processors operate synchronously as per the schedule pre-determined by the compiler, which avoids explicit synchronization. This mode is termed as synchronous MIMD and is most suited for instruction level parallelism.

### 3.1.4 Inter-processor Communication

Originally, communication of register contents between processors was only permitted via a shared external memory. This involves a large overhead in terms of clock cycles, since each processor has to request for access and a round-robin mechanism grants access. To enable quick exchange of register values between processors a multi-port register file consisting of 32 registers was introduced, which is accessible to all the processors simultaneously. This allows sharing of register values without having to alter the instruction set architecture.

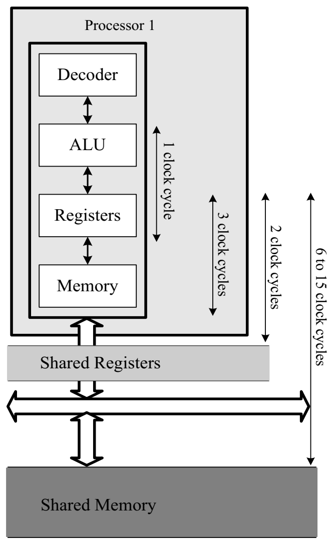

**Figure 3. Communication Hierarchy**

Further, since a single external memory is accessed by all the processors via a common shared bus, a bus contention is inevitable during simultaneous memory access. To circumvent this bottleneck, fast-memory-access was added where a single processors fetches data for all the processors in a single wide-word fetch (128-bit). This fetched data is then distributed internally among the processing elements (32-bits). This mode of operation bypasses the bus arbitration and contention incurred during shared data access. To summarize, the variation in terms of clock cycles observed within the processor hierarchy is shown in Figure 3. Communication can also be chosen a reconfigurable operating mode depending on the amount and frequency of data communication between processors.

## 3.2 Power Savings

Reduction in power when used in a multiprocessor mode, is as essential as achieving a significant speed up. The QuadroCore architecture by itself is optimized for both time and power. Low-power design strategies such as clock gating, dynamic power optimization have been incorporated in the synthesis design flow. Although these are methods of architectural modifications, the optimization of application dependent dynamic power remains unaddressed. Hence, a reconfigurable mode was introduced to save power. Here, switching between these two modes of operations is a two-step process. In the first step, the interconnect is reconfigured to allow broadcasting instructions (when switching from MIMD to SIMD) and this is accompanied by switching-off the instruction fetch and decode stages. Since the design uses multiple power rails, the power savings include both static and dynamic power.

## 4 Case Study: Multiplication in Binary Extension Fields

The polynomial basis representation is a common representation for the elements of a binary extension field $GF(2^m)$. The elements of $GF(2^m)$ can be expressed as a binary polynomial of degree at most $m-1$ as follows:

$$a(x) = \sum_{i=0}^{m-1} a_i \cdot x^i \text{ with } a_i \in \{0,1\} \tag{1}$$

We selected the binary extension field $GF(2^{233})$ for the analysis of the multiplication. This is one of the fields that has been suggested for ECC by the National Institute of Standards and Technology (NIST) [9]. In order to reduce the complexity of the polynomial multiplication, we apply the Karatsuba method [7]. Using the classical multiplication method, the three coefficients of the product $(a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + (a_1b_0 + a_0b_1)x + a_0b_0$ are computed with $4$ multiplications and $1$ addition from the four input coefficients $a_1$, $a_0$, $b_1$, and $b_0$. The Karatsuba method uses only 3 multiplications and $4$ additions:

$$\begin{aligned} &(a_1x + a_0)(b_1x + b_0) = \\ &a_1b_1x^2 + ((a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0)x + a_0b_0. \end{aligned} \tag{2}$$

By applying the Karatsuba method to larger polynomials the costs of extra additions is negligible compared to the saved multiplications. This way, the Karatsuba method achieves an asymptotical complexity of $O(m^{1.58})$ compared to the complexity of $O(m^2)$ for the classical method.

### 4.1 Implementation Scenario

The finite field multiplication in $GF(2^{233})$ is used to benchmark the QuadroCore architecture. The word width of our uni-processor comprises $w = 32$ bit. Therefore, the input polynomials $a(x)$ and $b(x)$ are divided into eight 32 bit words:

$$A(x) = \sum_{j=0}^{n-1} A_j \cdot X^{jw}, B(x) = \sum_{j=0}^{n-1} B_j \cdot X^{jw} \tag{3}$$

with $n = 8, w = 32$. The coefficients of higher degree than the considered binary field ($233 \cdots 255$) are padded with zeros. By applying the Karatsuba method iteratively, the multiplication of binary polynomials of degree 232 can be calculated with 27 finite field multiplications at word-level (see Table 2). The word-level multiplications are distributed to the four processors *PE1* $\cdots$ *PE4* of the QuadroCore architecture. In this way, always four partial products can be processed in parallel using SIMD mode. Here the instruction stream is the same for all the processors. For each partial word-level multiplication the processors calculate the sum of the words $j$ of the input polynomials $A(x)$ and $B(x)$. In binary fields, the sum of the input coefficients is easily calculated by an *XOR* operation. For example the fourth row of Table 2 shows that processor element 1 calculates the word-level multiplication

$$C = (A_0 + A_2) \cdot (B_0 + B_2). \tag{4}$$

The multiplication at word-level itself is performed using *shift-and-XOR* instructions [8]. The product $C$ will be a polynomial of double word length, which is stored in two registers containing the high *(H)* and low *(L)* word of the product, respectively. Finally, the partial products are added, i. e. *XORed*, to the corresponding word segments $c_i$ of the result (cf. Table 2).

**Table 2. Parallelized Karatsuba multiplication for QuadroCore**

| PE | sum of input words j | partial words of the resulting product | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | c15 | c14 | c13 | c12 | c11 | c10 | c9 | c8 | c7 | c6 | c5 | c4 | c3 | c2 | c1 | c0 |
| PE 1 | 0 | | | | | | | | H | LH | LH | LH | LH | LH | LH | LH | L |
| | 0,1 | | | | | | | | H | L | H | L | H | L | H | L | |
| | 1 | | | | | | | H | LH | LH | LH | LH | LH | LH | LH | L | |
| | 0,2 | | | | | | | | H | LH | L | | H | LH | L | | |
| | 0,1,2,3 | | | | | | | | H | L | | | H | L | | | |
| | 1,3 | | | | | | | H | LH | L | | H | LH | L | | | |
| PE 2 | 2 | | | | | | H | LH | LH | LH | LH | LH | LH | LH | L | | |
| | 2,3 | | | | | | H | L | H | L | H | L | H | L | | | |
| | 3 | | | | | H | LH | LH | LH | LH | LH | LH | LH | L | | | |
| | 0,4 | | | | | | | | H | LH | LH | LH | L | | | | |
| | 0,1,4,5 | | | | | | | | H | L | H | L | | | | | |
| | 1,5 | | | | | | | H | LH | LH | LH | L | | | | | |
| | 0,2,4,6 | | | | | | | | H | LH | L | | | | | | |
| PE 3 | 0,1,2,3,4,5,6,7 | | | | | | | | H | L | | | | | | | |
| | 1,3,5,7 | | | | | | | H | LH | L | | | | | | | |
| | 2,6 | | | | | | H | LH | LH | LH | L | | | | | | |
| | 2,3,6,7 | | | | | | H | L | H | L | | | | | | | |
| | 3,7 | | | | | H | LH | LH | LH | L | | | | | | | |
| | 4 | | | | H | LH | LH | LH | LH | LH | LH | LH | L | | | | |
| | 4,5 | | | | H | L | H | L | H | L | H | L | | | | | |
| PE 4 | 5 | | | H | LH | LH | LH | LH | LH | LH | LH | L | | | | | |
| | 4,6 | | | | H | LH | L | | H | LH | L | | | | | | |
| | 4,5,6,7 | | | | H | L | | | H | L | | | | | | | |
| | 5,7 | | | H | LH | L | | H | LH | L | | | | | | | |
| | 6 | | H | LH | LH | LH | LH | LH | LH | LH | L | | | | | | |
| | 6,7 | | H | L | H | L | H | L | H | L | | | | | | | |
| | 7 | H | LH | LH | LH | LH | LH | LH | LH | L | | | | | | | |

## 5 Results

Table 2 shows the distribution of the multiplications on the four processors (PE1, PE2, PE3 and PE4). The computed partial words by each of the processors are stored in the external shared memory to be accessible by all the other processors. Since the inter-processor communication is minimal, the asynchronous mode of operation was chosen. In the first case (asynchronous MIMD) all the processors operated independently on their own local instruction and data streams. In the MIMD-SIMD mode, the processors execute blocks of code in SIMD mode whenever possible. In case of data dependent variations to the control path, the mode is switched back to MIMD. The processors (PE2, PE3 and PE4) are reconfigured to operate based on the instruction stream of the 'master' (PE1), when executing the same function, e. g. the word-level multiplication. Each time, reconfiguring between modes consumes one clock cycle. Since all the processors need to execute the same instruction stream, all the processors need to be synchronized before entering the SIMD mode. Hence, a difference in the execution times between SIMD and MIMD modes. The reconfiguration functionality was hand-coded in assembly as a first proof of concept.

### 5.1 Timing Analysis

Table 3 compares the variation in execution time for the application in asynchronous MIMD mode and using MIMD$-$SIMD reconfigurable mode, for an operating frequency of 200MHz. A single processor implementation requires 9311 clock cycles, where as the multiprocessor implementation requires 3077 in the asynchronous MIMD mode and 3237 clock cycles in the SIMD$-$MIMD reconfigurable mode. The performance improvement and the power savings in the SIMD$-$MIMD mode confirms the advantage of reconfiguration.

In [12] a word-level based multiplication in the binary field $GF(2^{191})$ needs 15,344 clock cycles on a SPARC V8-compliant LEON-2 processor. Moreover, Elliptic Semiconductor Inc. claims to compute a finite field multiplication in $GF(2^{233})$ within 7600 clock cycles on a MIPS-based embedded processor [2]. Therefore, a reasonable performance is achieved with our QuadroCore architecture (cf. Table 3).

### 5.2 Power Analysis

The design composed of the four processors and their respective instruction and data memory, as shown in Figure 1 was synthesised with UMC's 90nm standard cell technology resulting in a total area of 4.23 sq mm. A 90nm memory (standard performance and low-K) was used for the local data and instruction memory. The final gate-level netlist was simulated and the switching activity was back annotated within PrimePower from Synopsys Inc., to determine the actual power values based on the application. A power savings of 23.4% was noted when used in SIMD mode, as compared to the MIMD mode. The resulting energy savings was 15.54%. From the detailed power reports for the QuadroCore architecture at 200 MHz and 1.0 V supply, it was seen that the total dynamic power is nearly 90% (62.64 mW) of the total power consumption (64.64 mW). It has to be noted that the dominant dynamic power is entirely ap-

**Table 3. Performance variations with Operating Mode**

| Operating Mode | Execution Cycles | Speedup | Power | Energy |
|---|---|---|---|---|
| Single Processor | 9311 cycles | 1 | 20.38 mW | 0.949 $\mu$J |
| Asynchronous MIMD | 3077 cycles | 3.03 | 64.64 mW | 0.994 $\mu$J |
| MIMD$-$SIMD | 3237 cycles | 2.88 | 49.51 mW | 0.801 $\mu$J |

plication dependent. Hence, power savings for an application can be achieved only by further reducing the dynamic power. Further, it was noted that about 80% of the total power was contributed by the on-chip memory. Among the rest (20%), the register file itself had a contribution of about 26%. As compared to the single processor implementation, the energy savings is higher for the multiprocessor implementation.

## 6 Conclusions

The proposed QuadroCore reconfigurable multiprocessor architecture adds minimal variations to the base multiprocessor architecture and allows runtime reconfiguration for power and energy savings, in addition to a speed-up. Overall, when comparing the reconfigurable multiprocessor to the fixed multiprocessor implementation, the maximum operating frequency was unaltered, but an area increase of 10% was observed. For the multiprocessor implementation of the finite field multiplication used in $GF(2^{233})$, a 3.03 times speed-up was observed as compared to a single processor implementation. Further, the first results indicate, power savings of 23.4% using the SIMD mode, as compared to the MIMD mode. This is achieved by reconfiguring the multiprocessor so that the unused parts are powered-off. This resulted in an energy savings of 15.54%. In particular, the proposed scheme of reconfiguration can also be extended to other processors. A further increase in speed-up could be achieved by optimizing the algorithmic partitioning, which would also result in higher energy savings.

Future work includes extension to incorporating reconfigurability to allow a wide ALU implementation with a word-length of 128-bits using the individual 32-bit ALUs, for improvement in the execution time of the multiplication algorithm.

## Acknowledgement

The research described in this paper was funded in part by the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung – BMBF), registered there under grant number 01AK065F (NGN-PlaNetS), and by the German Research Foundation (Deutsche Forschungsgemeinschaft – DFG) under project RU 477/8 and SFB 614.

## References

[1] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 2000.

[2] Elliptic Semiconductor Inc. *CLP-22: Elliptic Curve Point Multiplier Core*, 2007. Available from http://www.ellipticsemi.com.

[3] M. Gruenewald, U. Kastens, D. K. Le, J.-C. Niemann, M. Porrmann, U. Rueckert, M. Thies, and A. Slowik. Network application driven instruction set extensions for embedded processing clusters. In *PARELEC 2004, International Conference on Parallel Computing in Electrical Engineering, Dresden, Germany*, pages 209–214, 7 - 10 Sept. 2004.

[4] T. R. Halfhill. *Ambric's New Parallel Processor*, Oct. 2006. Available from http://www.ambric.com.

[5] D. Hankerson, J. L. Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–24, London, UK, 2000. Springer-Verlag.

[6] M. Hussmann, M. Thies, U. Kastens, M. Purnaprajna, M. Porrmann, and U. Rueckert. Compiler-driven reconfiguration of multiprocessors. *in Proceedings of the Workshop on Application Specific Processors (WASP) 2007 held in conjunction with the Embedded Systems Week, 2007 (CODES+ISSS, EMSOFT, and CASES)*, pages 3–10, 2007.

[7] A. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[8] C. K. Koc and T. Acar. Montgomery multiplication in GF($2^k$). *Des. Codes Cryptography*, 14(1):57–69, 1998.

[9] National Institute of Standards and Technology (NIST). *Digital Signature Standard (DSS)*, volume FIPS 186-2, chapter Recommended elliptic curves for federal government use, pages 24–48. U.S. Department Of Commerce, 27 Jan. 2000.

[10] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *IEEE Micro*, 23(6):46–51, 2003.

[11] Tilera Corporation. *Tilera64 Processor Family*, Aug. 2007. Available from http://www.tilera.com.

[12] S. Tillich and J. Grossschaedl. A simple architectural enhancement for fast and flexible elliptic curve cryptography over binary finite fields GF($2^m$). In *Advances in Computer Systems Architecture*, Lecture Notes in Computer Science, pages 282–295. Springer Verlag, 2004.