C. GRABBE, M. BEDNARA, J. SHOKROLLAHI, J. TEICH & J. VON ZUR GATHEN (2003). A High Performance VLIW Processor for Finite Field Arithmetic. In *Proc. of The 10th Reconfigurable Architectures Workshop (RAW-03)*.



# A High Performance VLIW Processor for Finite Field Arithmetic

C. Grabbe, M. Bednara, J. von zur Gathen, J. Shokrollahi, J. Teich
University of Paderborn, Paderborn, Germany
{grabbe, bednara, teich}@date.upb.de, {gathen, jamshid}@upb.de

## Abstract

*Finite field arithmetic forms the mathematical basis for a variety of applications from the area of cryptography and coding. For finite fields of large extension degrees (as for cryptography), arithmetic operations are computation intensive and require dedicated hardware support under given timing constraints. We present a new architecture of a high performance VLIW processor that can perform basic field operations in parallel as well as complex instructions as needed for elliptic curve cryptography. The control path is microcoded, so the instruction set can easily be modified or extended. The modular data path structure along with an FPGA-optimized design facilitate adaption to various resource and timing requirements.*

## 1 Introduction

Algorithms from the areas of cryptography and coding make excessive use of finite field arithmetic operations. Due to fundamental differences in finite field and integer arithmetic, pure software solutions are usually inefficient since general purpose processors are designed for integer arithmetic. This holds especially for fields with large prime extensions.

In [1] we have presented a modular processor architecture for elliptic curve operations based on the finite field $GF(2^{191})$ which was designed using bit serial multipliers. In this work, we present a new processor architecture that is designed as coprocessor for efficiently computing finite field operations in $GF(2^{233})$. (This finite field is advised by NIST for future designs of elliptic curve algorithms and regarded to be secure for next years.) The main characteristics of the new processor design are

- Functional data path units can be used independently and in parallel (VLIW architecture)
- A single pipelined parallel hybrid-Karatsuba multiplier that performs a multiplication in $GF(2^{233})$ in 9 clock cycles
- A large crossbar switch as flexible interconnection structure between all functional units
- Hierarchical microcoded control path with VLIW-style low-level instructions as well as complex elliptic curve instructions

The processor design is optimized for FPGA designs, especially the control path makes use of dedicated on-chip memories of modern FPGAs (for our design, we used a Xilinx Virtex-II-E FPGA). The data-path design is kept modular and can be adapted to various timing and area constraints, since the implementation of finite field operations allows for many degrees of freedom (see Section 3). Furthermore, the microcoded control path can easily be extended with new instructions.

The main operation in elliptic curve cryptography application is the operation $k \times P$, i.e. the multiplication of a curve point $P$ with a large integer $k$ (in our case, k is a 233bit integer). This operation is much more complex than a normal integer multiplication and is performed on different abstraction levels. In contrast to other approaches, which allow the programmer just to access the highest (algorithmic) level, our processor provides an interface to each abstraction level. That means, on the algorithmic level, we provide a system call of the form *curve_mult (param)* which performs a complete $k \times P$ operation. On the curve level, the basic curve operations can be accessed via the *curve_add (param)*, *curve_double (param)* and *invert_point (param)* system calls. On the finite field level, the user can access low-level field operations provided by system calls as *gf_add (param)*, *gf_square (param)* etc.

For the low-level field operations we use VLIW instruction words since the data path can perform up to three different field operations in parallel (except for field inversion which is decomposed into a sequence of other field operations). All instructions of the higher levels use complex non-VLIW instruction codes that are translated into a sequence of basic

This work has been supported by DFG Sonderforschungsbereich 376 "Massive Parallelität."

field operations by the internal microcoded control path.

The paper is organized as follows. Section 2 discusses some related work. In Section 3 we give a very short introduction into the ideas of elliptic curve cryptography. In Section 4 we show the architecture of our processor in detail, while Section 5 describes the instruction set. In Section 6 we give an overview of the current project status and some performance results. Section 7 is a summary.

## 2. Related work

There are some other works concerning the programmable or microcoded implementations of elliptic curve coprocessors. Three of these works are [6], [4], and [3]. [6] uses the normal basis to represent the finite field. It uses the Massey-Omura multiplier with $n$ clock cycles to multiply two elements of $GF(2^m)$. [4] uses triangular basis and [3] the polynomial basis. All of these implementations are based on small area implementations and use all of the arithmetic modules in a iterative manner.

The goal in our implementation is using the resources of large FPGAs for the best performance. To achieve this, our design uses a fast multiplier which performs the total multiplication in a few clock cycles. This multiplier and other arithmetic units can be used in parallel to achieve the best performance. Generally, the whole processor design is optimized to exploit the inherent parallelism of the elliptic curve multiplication algorithm. To our knowledge, this is the first FPGA based parallel processor for 233bit operands.

## 3. Elliptic Curve Cryptography

This section gives a short introduction into the application of elliptic curves in the area of cryptography [2].
The points of an elliptic curve defined over a finite field form a finite group, and the group operation is point addition. The basic operation in elliptic curve cryptosystems is the computation of $m\mathcal{P}$, where $\mathcal{P}$ is a point and $m$ a (large) integer. The computation of $m\mathcal{P}$ is done as a sequence of repeated point additions and doublings. Elliptic curve cryptosystems (ECCs) rely on the fact that solving the discrete logarithm problem on an elliptic curve is a hard task. That means, for a given $\mathcal{P}$ and $m$, computing $m\mathcal{P}$ is of polynomial complexity, but computing $m$ from only $\mathcal{P}$ and $m\mathcal{P}$ is, in general, assumed to be infeasible in polynomial time [7]. Some care has to be taken in order to avoid special curves with easy discrete logarithms. For a field of characteristic two, the minimum number of bits required to represent the finite field elements is recommended to be larger than 160 to resist "generic" attacks. ECCs defined over such fields are assumed to be as secure as RSA systems with 1024 bits [2]. The short keys make elliptic curve cryptosystems attractive in communication systems with tight bandwidth limitations. Fig. 1 shows that the point multiplication naturally decomposes into a hierarchy of three levels. The upper levels use essentially the subroutines provided by the lower levels. Each level can be optimized in order to meet the given area/performance constraints.

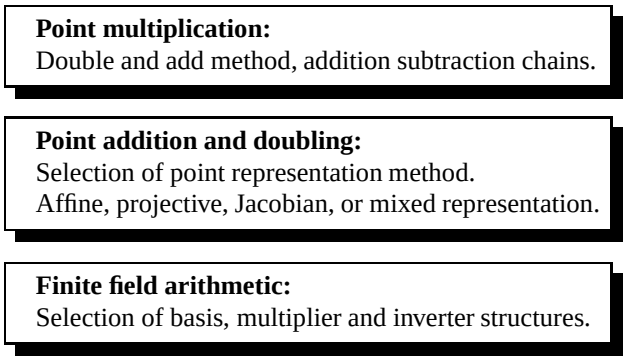


**Figure 1. Hierarchical levels of elliptic curve point multiplication.**

There are several ways to implement the two topmost levels of the hierarchy. There are several point representations, which can influence the performance of point addition and doubling. Windowing methods based on precomputation can also be used to improve point multiplication performance. A hardware which is required to use all of these possibilities, should have a flexible structure. Such a flexibility can be achieved using a microcoded architecture.

## 4. Processor Architecture

An architectural overview of the processor design is given in Fig. 2. The host processor communication is done via two memories. The program memory holds the sequence of instruction words to be executed. Each instruction word is 64 bits wide, independently from its type (VLIW/non-VLIW).
The operand memory holds all parameters necessary for execution, i.e. the finite field elements for the low-level operations and the curve parameters for the elliptic curve operations, respectively.
Both memories can be accessed by the host processor via memory mapped i/o or i/o-channel, depending on the host processor system.

### 4.1 Control Path Architecture

The control path (Fig. 3) consists of two subunits:

- **(1) level_0_ctrl** is responsible for instruction fetching, parts of the instruction decoding and program memory

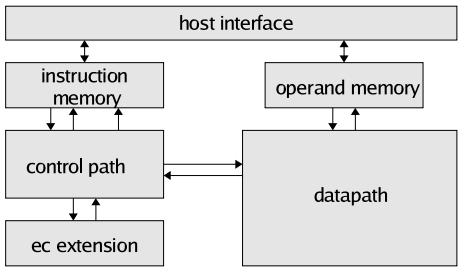


**Figure 2. Processor architecture**

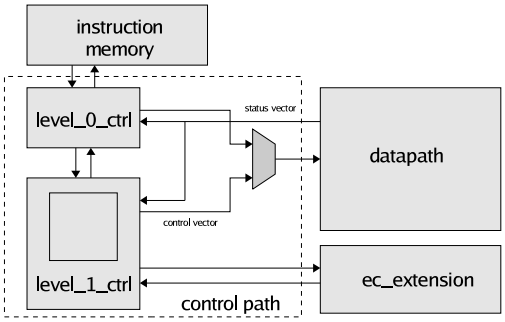


**Figure 3. Control path architecture**

addressing. All non-arithmetic instructions like jump and host data transfer are handled by level_0_ctrl. Each instruction fetched is analyzed with respect to its type (data transfer, VLIW arithmetic, curve arithmetic) and forwarded to the appropriate unit.

- **(2) level_1_ctrl** is a microcoded control unit that translates all instructions for curve arithmetic and finite field inversion into a sequence of micro instructions. The complex instruction set of the processor can be extended by modifying the level_1_ctrl microcode.

Generally, the code sequences stored in the microcode memory could also be executed from the program memory. In this case, each micro instruction becomes a VLIW instruction. Decoding of these instructions, however, requires some clock cycles while the micro instructions do not need to be decoded. So, for long instruction sequences (as resulting from the curve operations) are executed much more efficient as microcode than from the instruction memory.
On the algorithmic level of the curve multiplication, we use the double&add method, where each bit of the integer $k$ determines the curve operations that has to be performed in a certain step of the computation. An additional unit called ec_extension holds the integer $k$ and allows for a subsequent examination of all bits of $k$.

## 4.2 Data path Architecture

The data path of our processor consists of 3 arithmetic units:

- A hybrid Karatsuba multiplier unit that multiplies two field elements. A complete multiplication requires 9 clock cycles and the multiplier has a pipeline rate of 2, i.e. each two clock cycles new operands can passed to the multiplier, so a single input port for both operands is sufficient.
- An adder unit which performs a complete addition of either two or three operands in one clock cycle
- A squarer unit that performs a squaring in one clock cycle

Furthermore, we have two independently addressable register files (rfA and rfB) with four 233bit registers each. One of the rfA registers is connected to a comparator that generates a flag if the operand stored in this register is zero.

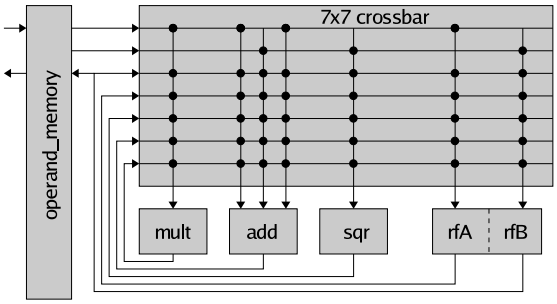


**Figure 4. Data path architecture**

As shown in Fig. 4 these units are interconnected via a 7x7 unidirectional crossbar switch that can connect the output of each data path unit to the input of any other (or the same) data path unit in parallel. The crossbar is also connected to the read interfaces (memA and memB) of the operand memory. This memory stores 64 operands of 233bit each, so the crossbar can read two operands from the memory and pass directly to any data path unit. For simplifying the crossbar design, we restrict that each input of an arithmetic unit or register file can be connected only to one memory interface instead of both. With a proper operand memory addressing scheme, this restriction does not result in a performance loss. Write access to the memory is possible only via a single write port that is connected to the output of register file rfB (the second write port of the operand memory is reserved for host communication).

Each arithmetic unit contains an output register that stores the result of the operation as long as no new operands are loaded (for adder and squarer) or for two clock cycles (for the multiplier). Thus, operands for any arithmetic unit can be read directly from another (or the same) arithmetic unit without storing the result in the register file (which can be done in parallel, however). [1]
The data path is controlled via a 50bit control vector generated by the control path. The control vector is described in Section 5.

## 5. Instruction Set

The instruction set of the processor is divided into 3 groups:

- control instructions like jmp and host communication instructions
- complex instructions for curve operations and field inversion
- low-level VLIW instructions

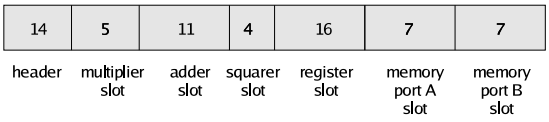


**Figure 5. Instruction word**

### 5.1 VLIW Instructions

For the VLIW instructions, each instruction word consists of a number of so called slots (Fig. 5), where each slot is assigned to a data path unit. As an example, the adder slot consists of the following bits:

- doa (do addition): indicates that an addition has to be done
- addmode: selects the number of addition operands (2 or 3)
- add_cba, add_cbb, add_cbc: crossbar addresses for the operands A, B, and C. Each address consists of 3 bits and selects one data source for the adder. If only 2 operands are needed, add_cbc is ignored.

[1] It should be noted that it is in the programmer's responsibility to proceed with the results of arithmetic units as long as they are available in the result register, since they will be overwritten when the next result becomes valid.

The slots for the multiplier and squarer are similar, while the memory and register file slots consist of memory/register addresses, crossbar addresses and write enable bits.

**Example 5.1.** *Consider an addition of 3 operands, with one operand from memory port B, one from register file rfA, and one from squarer output register. The doa bit must be set in order do perform an addition, the addmode bit must be set accordingly for 3 operands. All crossbar addresses must be set to select the operand sources.*
*Furthermore, in the rfA slot the address of the required operand register must be set as well as the port B address in the memory slot.* [2] *The VLIW assembler instruction would be*

```
add memb, rfa, sqr; rfa(3); memb(0x1C);
```

*The* `add` *opcode indicates an addition instruction, the three operand mnemonics give the crossbar addresses, and the rfa and memb keywords give the register and memory addresses, respectively. Additionally to the add instruction, the same instruction word could also contain one multiplication, one squaring and two load/store operations. Here, the according slots are not used and the assigned data path units perform a NOP operation.*

The header slot is required by level_0_ctrl for instruction decoding. All bits except for the header are used directly as control vector for the data path without further decoding.
The data path structure allows for the execution of one VLIW instruction per clock cycle, but it must be taken into account that a multiplication requires 9 clock cycles. During the multiplier latency, of course, other VLIW instructions (also multiplications) can be started. Furthermore it must be taken into account that the multiplier has only one input which requires the operands to be loaded serially. The first step of a multiplication is loading the first operand, the second step is loading the second operand and triggering the multiplication.

### 5.2 Complex Instructions

The complex instructions for curve operations are

- `ec_mult` (curve multiplication)
- `ec_add` (curve point addition)
- `ec_dbl` (curve point doubling)
- `ec_inv` (curve point inversion)

[2] Actually, the memory address must be set in the previous instruction word due to a 1-clock cycle output latency of the operand memory.

Additionally, we have a complex instruction `gf_inv` for the inversion of a finite field element. Although this is a finite field operation, it is so complex that it must be realized as a level_1_ctrl microprogram.
Curve instructions expect the required operand sets in a fixed order of addresses in the operand memory. The address space of the operand memory is sufficient to hold two complete parameter sets for a `ec_mult` instruction. Thus, the only argument to `ec_mult` is the parameter set base address, which results in a much simpler instruction word, most of the bits are not used. The `ec_add` and `ec_dbl` also require a base address as single parameter.

## 6. Project Status and Performance Results

Currently, the data path and operand memory of our processor is completely simulated and synthesized for a Xilinx XC2V6000 FPGA. With a slightly timing optimization and placement constraints, we could achieve a clock period of about 100MHz. The design requires 19440 LUTs (28%) and 16970 slice flipflops (25%). The control path is a VHDL simulation model which is currently being optimized for FPGA synthesis. Since the control path complexity is much less than the data path complexity, we expect lower delays and thus a clock frequency of about 100 MHz (limited by the data path) for the complete design.
The host interface highly depends on the host processor system. For prototyping purposes, we use a very simple interface connecting the coprocessor to a Texas Instrument C6701 DSP via an 8 bit serial interface. This interface allows not for memory mapped i/o.
We will finish the complete implementation of the processor design including the micro code for the final version of this paper.

### 6.1 Elliptic Curve Performance

On the algorithmic level of the elliptic curve $k \times P$ operation, we use the double&add method and the Affine/Jacobian hybrid coordinate representation (For an analysis of different hybrid coordinate representations see [1]).

The data flow graph for an elliptic curve point doubling operation is shown in Fig. 6. The inputs $x$, $y$ and $a$ are curve parameters, $m$ and $M$ denote a multiplication, $A$ an addition and $S$ a squaring. Executed by the level_2_control unit, the complete point doubling operation requires 31 clock cycles, while a complete point addition operation (which is much more complex) requires about 42 clock cycles. The double&add method subsequently evaluates all bits of the integer $k$ and performs a point doubling operation in each step and a point addition if the currently evaluated bit of $k$ is one. In the average case, half of the bits of $k$ are one, so we

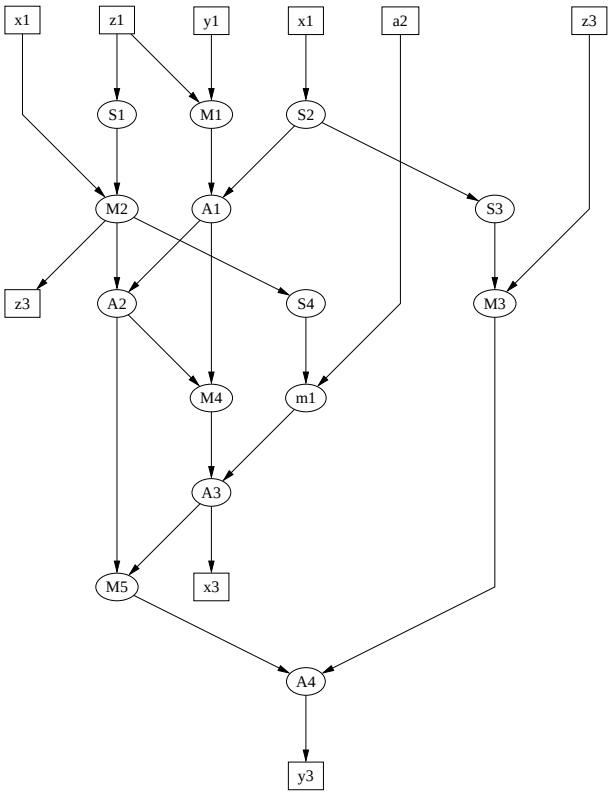


**Figure 6. Data flow graph of an elliptic curve point doubling operation**

have a total of 233 point doublings and 117 point additions. Assuming a 100MHz clock rate, this requires $120\mu sec$ with our processor. At the end of the complete $k \times P$ operation, a coordinate transformation of the point coordinates back to the affine representation is necessary. This requires a finite field inversion which is the most complicated field operation and thus realized as a level_2_ctrl microprogram.
For field inversion, we use the algorithm presented in [5] which is based on addition chains. This algorithm requires 15 subsequent field multiplications which are all dependent, thus we can not exploit the pipelining structure of the Karatsuba multiplier. This causes additional time of $1.35\mu sec$.
To our knowledge, this is the first FPGA based field arithmetic processor for 233bit operands, to a comparison of these results to other approaches is currently difficult.

## 7. Summary

We have presented a new high performance processor architecture for finite field operations in $GF(2^{233})$ and elliptic curve operations used for cryptography. Since these operations are very computation intensive, our goal was the

exploit the inherent parallelism of these algorithms as far as possible. So we have designed a VLIW-style processor with a data path that can compute up to 3 different finite field operations in parallel. We use a very fast hybrid Karatsuba multiplier for field multiplication and a large crossbar switch for interconnecting the data path units. The hierarchical control path is microcoded and can execute low-level VLIW operations as well as complex elliptic curve operations. Changing the microcode of the control path can be easily extended by new instructions.
We have implemented parts of the design on a VIRTEX-II FPGA and achieved a clock rate of about 100MHz allowing for a complete 233bit elliptic curve multiplication in less than $130\mu sec$.
In contrast to other processors, we provide the programmer with functions for the high levels of elliptic curve algorithms as well as for the low-level finite field operations. So, the processor can also be used for efficient implementations of other finite field based algorithms from the area of cryptography or coding theory.

## References


[1] M. Bednara, M. Daldrup, J. Shokrollahi, J. Teich, and J. von zur Gathen. Reconfigurable implementation of elliptic curve crypto algorithms. In *Proc. of The 9th Reconfigurable Architectures Workshop (RAW-02)*, Fort Lauderdale, Florida, U.S.A., April 2002.

[2] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*. Number 265 in London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.

[3] James Ross Goodman. *Energy Scalable Reconfigurable Cryptographic Hardware for Portable Applications*. PhD thesis, Massachusetts Institute of Technology, August 2000.

[4] M.A. Hasan and A.G. Wassal. Vlsi algorithms, architectures, and implementation of a versatile $GF(2^m)$ processor. *IEEE-TC*, 47(10):1064–1073, October 2000.

[5] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.

[6] K.H. Leung, K.W. Ma, W.K. Wong, and P.H.W. Leong. FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 68–76, Napa Valley, California U.S.A., 2000.

[7] Alfred J. Menezes, Ian F. Blake, XuHong Gao, Ronald C. Mullin, Scott A. Vanstone, and Tomik Yaghoobian. *Applications of finite fields*. Kluwer Academic Publishers, Norwell MA, 1993.