A 16-Bit Microprocessor Chip for Cryptographic Operations on Low-Resource Devices

Erich Wenger[1], Martin Feldhofer[1], Norbert Felber[2]

[1]Institute for Applied Information Processing and Communications, Graz University of Technology

[2]Integrated Systems Laboratory, Swiss Federal Institute of Technology Zürich

{erich.wenger,martin.feldhofer}@iaik.tugraz.at,felber@iis.ee.ethz.ch

Abstract

Security services in embedded systems and contactless devices (smart cards, RFID tags) get more and more important. In this paper we investigate the design and implementation of a 16-bit general-purpose microprocessor called Neptun. The purpose of this design is to have a platform for the evaluation of the Elliptic Curve Digital Signature Algorithm (ECDSA) using the NIST P-192 parameters. The processor efficiently uses the available chip area and performs an ECDSA signature generation, using only 1405K clock cycles. For a signature verification 2675k clock cycles are required. This fast computation time is possible through the use of a Harvard architecture, a self-designed CPU core and a self-designed instruction set. With the 5K program memory and a clock frequency of 55.5 MHz, the design can also be used as a security-extension for arbitrary devices. In addition to a working prototype chip, we have a full tool chain that includes an instruction-set simulator and an assembler for producing executables.

1 Introduction

The two technologies of RFID tags and smart cards are on the brink of conversion. Whereas RFID tags have very serious power constraints, smart cards make use of powerful processors to calculate cryptographic algorithms. In the course of the 'Internet of things', those technologies ought to be combined to bring strong cryptography to RFID tags. An algorithm such as the Elliptic Curve Digital Signature Algorithm (ECDSA) is perfectly suited for such small implementations. Most importantly it requires a smaller data memory than an algorithm such as the Digital Signature Algorithm (DSA). ECDSA is used to sign a message or challenge using its private key. This signature can be used in the course of entity authentication. The central element within ECDSA is an elliptic curve point multiplication. For that efficient multiplication methods are required.

Instead of designing another cryptography-coprocessor, as it has been done by [3, 4, 6, 10, 11, 12], we decided to create a dedicated microprocessors. But what is the use of another processor platform, when there are already so many different architectures and designs out there? Our processor is specially designed to perform an ECDSA using an underlying prime field. This means that any registers, logic elements and memory entries that are not required for the calculation of ECDSA are left out. The result is a very small, area-optimized processor which has been published in [13]. With this paper, we want to show that our processor can be used as a general purpose processor and also be used for other algorithms and protocols. This new design, which we named Neptun[1], uses the same CPU as the area-reduced design, but now it is improved and more re-usable. It has been extended with additional data memory, a re-programmable program memory and several peripheral units. Nevertheless we kept the original low-area requirement in mind and as a result, the processor still fits within one square millimeter.

The idea behind Neptun has been to make a platform which can be used to evaluate various algorithm implementations concerning their simple and differential power analysis resistance. Because Neptun comes with several I/O interfaces, it can be used to handle wireless communication protocols such as ISO-14443 or ISO-15693 [8, 9]. Neptun can also be used to efficiently add cryptography to arbitrary devices. To implement and verify complex algorithms such as ECDSA or the previously mentioned protocols, Neptun comes with a very advanced development environment written in Java. This software combines the functionality of a simulator and an assembler. Using those tools, the development time of programs for Neptun can be reduced significantly.

This paper is composed to present the used hardware architecture and the applicable tools. Section 2 gives a top-level view of Neptun. The Subsections 2.1, 2.2 and 2.3 explain how the CPU, the ALU and the instruction set are designed in order to achieve a very low runtime and keep the required chip area low. The tools for the efficient development of algorithms are presented in Section 3. Results concerning Neptun are shown in Section 4. Finally, conclusions are drawn in Section 5.

2 Hardware Architecture

Our processor has been primarily optimized for a small area requirement and secondly for performance. The security concerns are addressed on the algorithm level. The low area is achieved by using a small set of registers and a 16-

[1]Neptun is not an acronym. It simply is the planet with the fewest hits at google.com.

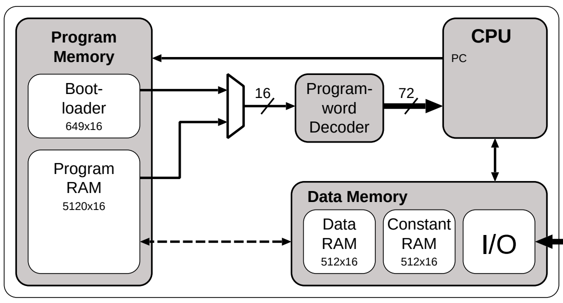

Figure 1. Top-level processor implementation.

bit word size. The efficient runtime is due to the optimized memory access and the used algorithms.

The processor design as it is shown in Figure 1 uses a modified Harvard architecture with a separated data and program memory.

The data memory is split into a data RAM, a constant RAM, some memory mapped I/O and a memory mapped program RAM. The data RAM is used for general purpose data. Such data can be the heap or the stack. The constant RAM is used to store parameters that are known at compile time. For memory-intensive algorithms, such as ECDSA, it is of advantage to have those constant parameters stored via a memory-mapped interface. We use a RAM macro because neither a Flash nor an EEPROM are available in the UMC-L180 target technology.

For the reusable processor design, the program memory has been split into a bootloader and a program RAM. The bootloader is implemented as a synthesized 16-bit look-up table. After power-up the bootloader initializes the program RAM with an executable, received via a serial or SPI interface (SPI Flash). For that purpose the program memory is also mapped within the data memory. It is important to mention that the program RAM can only be written during the execution of the bootloader. Like the constant RAM, program RAM is implemented as a RAM macro. Those RAM macros are much more area efficient than synthesized RAM blocks.

The central element used for calculation is the CPU. It contains the program counter which is used as an index for the program memory. The result of the program memory, also known as the program word is then decoded to serve as a control input vector for the CPU and the data memory. It is important to note that the bootloader uses the current value of the program counter as index, whereas the synchronous program RAM uses the next value (the input of the program counter register) as address input. We do not need to care about branch prediction because the design does not make use of a pipeline and branching is done within one cycle.

2.1 CPU

The important interfaces of the central processing unit (CPU) in Figure 2 are the two program counter states (current *PCxDP* and next *PCxDN*) and the memory interface.

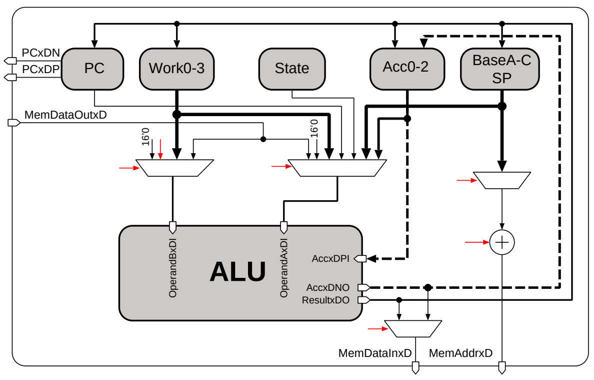

Figure 2. Design of the CPU. Two registers are selected with the two multiplexers and handled within the ALU. The result is then stored in the RAM or a register.

This consists of a data input and output and an address output. The chip select and write enable signals are directly encoded within the control vector and simply forwarded to the data memory module.

Contrary to AVR processors that have many general purpose registers, we tried to keep the number of the needed registers as small as possible. The focus has been on having as few registers as possible and all of those registers should have a special purpose. This design directive should contribute to the small area requirement. Storing data in the RAM instead of in registers has no disadvantage because memory access is also done within one clock cycle.

The two major parts of the CPU are the ALU and the set of registers. Those registers are described in the following:

Program counter. The program counter (PC) marks the current position of the currently executed instruction. Usually the PC is always incremented by one. There are a few operations (Jump, Call, Branch, ...) that can modify the PC differently. For such operations the ALU is used to manipulate of the PC in order to avoid any necessary additional hardware.

Stack pointer. During the execution of functions a stack can be used to store return addresses, function arguments or temporary values. Essentially the stack pointer is identical to the base registers with the difference of a special label.

Base pointer. Because elliptic curve field additions and multiplications need two source addresses and a destination parameter, three base pointer registers (BaseA - BaseC) are needed for memory-address generation.

Accumulator. A very important part for the multiplication of big integers is the multiply accumulate unit embedded within the ALU. The result of a multiplication has 32 bits. So in order to add some products, three 16-bit registers are required.

State. In this register, all the status flags of the processor are stored. These status flags are: Carry, Zero, Overflow and Negative.

Work registers. Unlike all the previously described registers, the work registers have no special purpose. In the course of an investigation of the requirement for ECDSA, four general purpose registers (Work0-3) have been defined. Those are the only registers that can be used as ALU operand A and B.

A way to reduce the required area is to use small multiplexer to select the two operands for the ALU. This is why the multiplexer of operand B can only select the work registers, a constant from the control vector and data from the previous memory access. Operand A on the other hand can access all registers and thus keeps the CPU flexible.

A very important part of the whole design is the way the signal *MemDataOutxD* is handled. This signal represents the data that has been read during a previous memory access. This data can be accessed and processed directly by the ALU and does not need to be stored in a register beforehand. This scheme reduces the time needed for a memory operation from two cycles to one cycle and makes our design special compared to other solutions.

Another modification concerning the optimized memory access is in connection with the result of the ALU. This data can be directly stored as a new result to the data memory. For memory-intensive algorithms such as the ECDSA, such an optimization speeds up certain operations like field multiplications and additions. Additionally it reduces the number of required temporary registers.

2.2 ALU

The most important part of the arithmetic logic unit (ALU), as it is shown in Figure 3, is the multiply-accumulate unit. Because of its size it is specially considered for power optimization. We simply use a method called operand isolation on the two input factors. This method should reduce the total energy requirement significantly. The 32-bit result of the multiplier can be used directly or added to the accumulator. The investigation of the NIST P-256 elliptic-curve parameters show the need to add/subtract values to/from the accumulator unit. The logic XOR is used to invert *OperandAxD*. As a result *OperandAxD* can be added or subtracted without the use of any extra major logic elements.

The other functionality of the ALU is similar to the functionality provided by most embedded processors currently available. The two major inputs of the ALU are the operands A & B, which has already been discussed in the previous section. In every cycle, they are added, ORed, XORed, ANDed and shifted. Because those logic blocks are fairly small they have not (yet) been considered for power optimizations. Only one of the intermediate results is selected and passed on to *ResultxDO*. By definition the subtrahend is always *OperandBxDI*. So only operand B needs to be invertible. The implementation of the ECDSA showed that this extra constraint on the operand selection is hardly noticeable.

The method for the update of the status flags is neglected for the focus of this paper. Nevertheless the implementation of the branching logic is worth some deeper investigation.

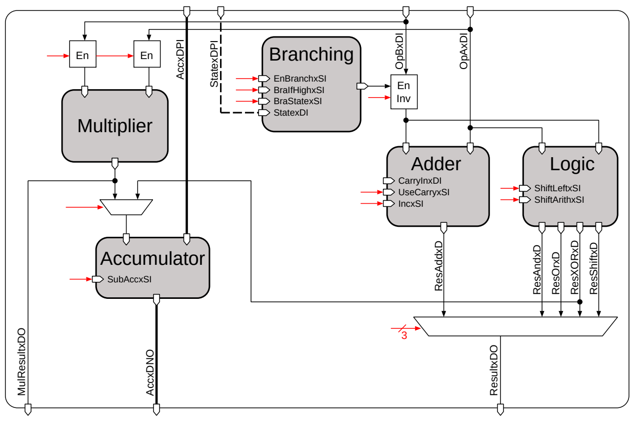

Figure 3. The ALU is capable of adding, subtracting, multiplying, accumulating, logic functions and branching.

Figure 4 shows a very efficient implementation of a branching logic. Operand A is the program counter. Operand B provides a relative jump address. The result is a new value for the program counter.

There are two possible cases to consider:

The branch condition is false. In this case operand B is forced to zero and the result is $PC + 1$. The one is added by setting the increment input of the adder to one.

The branch condition is true. Operand B is represented as a two's complement number. As a result $PC + OperandB + 1$ can result into a jump to a lower and higher program address.

The efficiency of this implementation is that the conditional jump is always executed in exactly one cycle. This can only be achieved because there is no pipeline used within the whole design. In general, every instruction except load immediate (**LDI**), which requires two cycles, needs one clock cycle only.

2.3 Instruction Set

Designing an instruction set requires a balancing act between performance, reusability and a small word size. On the one hand side, the instructions should be as flexible as possible, so they can be used efficiently for as many use cases as possible. This would suggest to use a big instruction word. On the other hand using a big instruction word means that a larger program RAM is required and this is contradicting the low-area design requirement.

We have used the well established Thumb [1] and AVR [2] instruction sets as reference because both of them use very efficient 16-bit instruction words. A 16-bit instruction word fits nicely to the 16-bit word size of our processor. Unfortunately, neither of the two instructions

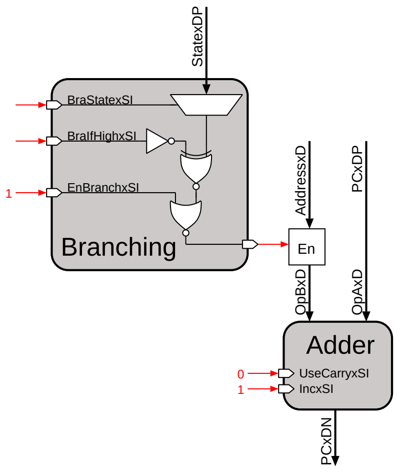

Figure 4. Dependent on the branch logic, either 1 or $OperandB + 1$ is added to the program counter.

sets fit the ECDSA algorithm perfectly. Our focus is on efficiently implementing ECDSA, so we introduced several optimizations for our instruction set:

- Prime field operations make heavy use of arithmetic instructions (additions, subtractions, multiplications). So for these instructions our instruction set is focused on flexibility.
- Several arithmetic instructions have been combined with data memory read and store instructions. Those instructions have a lot of operands (base register, immediate offset, two source registers) and as a result the flexibility of those instructions needs to be limited.
- Logic operations are mostly needed for the performance irrelevant calculation of the SHA-1 hash function, which is a small part of the ECDSA calculation. In order to have more entries in the instruction table available for the parallelized arithmetic instructions their operands are very limited.

2.4 Design for Test

For every chip which is going to be mass produced, it is important to think about testing. Our chip test is split into two phases. During the first phase, the RAMs are isolated using a method called block isolation which is shown in Figure 5. In this phase all registers are interconnected to a scan-chain which is used to test the registers and combinatory logic. The signal *TestModexTI* is set to one. As a result, all inputs of the block isolated RAM can be read and the output can be set via the scan chain. Because the bootloader is a synthesized look-up table it is also tested during the scan-chain tests. The bootloader is then used in the second phase to test the three RAM macros. Testing the RAM macros using the scan-chain would be very inefficient. The test time would increase significantly.

Using a scan-chain in a security related product is very critical. For the final product the scan chain must be disabled using some (one time writable) flags or completely avoided using a build-in self test for the security-critical parts.

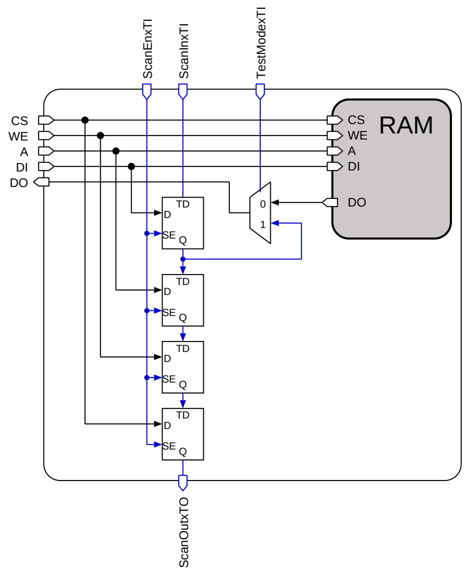

Figure 5. Block-isolation of a one-bit RAM.

3 Design Environment

In order to use an embedded processor efficiently, it is always important to have adequate and powerful tools. For our designed processor we wrote our own instruction-set simulator and assembler using Java:

Verification. This simulator is capable of debugging and simulating all the registers and memories as well as the memory mapped peripheral interfaces. So at an early stage of the design flow the program and the consistency between the simulator and the VHDL model can be verified. Because the simulator is written in the object-oriented language Java, it is easy to implement unit tests using a tool such as JUnit. The implemented ECC algorithms can be tested using a cryptographic library as a golden model.

Evaluation. A common criteria for the evaluation of cryptographic algorithms is their runtime. Because the processor frequency is not important for a logic simulation, the runtime is evaluated in cycles. The simulator is capable of generating various statistics such as: the runtime of every function, the number of times a function is called, the number of times a certain instruction is executed, the number of code lines per function, etc.

Debugging. Nobody writes correct code right from the start. Everybody makes a mistake at some point in the implementation of an algorithm. In such a case it is important to have debugging capabilities. Our design framework can trace the contents of all registers after each cycle or the content of certain RAM addresses at important points of the algorithms. Also a live comparison between a golden model and the actual assembler implementation can be performed and has already been implemented.

Assembler. The assembler is used to parse source code and generates executables plus various other data structures. Usually assembler is written in dedicated files (with .asm extensions). Our assembler code is written within Java files. This gives the advantage that an editor (e.g. Netbeans) can be used for syntax highlighting and several spell checks are already done by Java itself.

For the design of the processor it was important to more than generate binary executables. Other formats such as data formatted in S-records (for program RAM) or VHDL tables (for bootloader synthesis) are also required.

4 Results

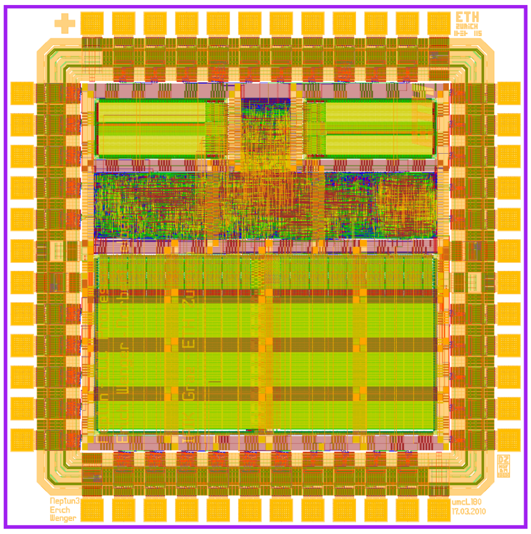

Figure 6. The final layout of Neptun.

The microprocessor design called Neptun, which has been described so far, has been synthesized, placed, routed, power simulated, produced and successfully tested. As manufacturing process of the chip, we used the 180 nm technology from UMC. The final layout is visible in Figure 6. The inner dimensions without the power rings are 1 mm x 1 mm (1 mm^2, 106 kGE). The outer dimensions of the scribe lines are 1.55 mm x 1.55 mm (2.4 mm^2). Because the inner dimensions have been fixed beforehand, the focus has been on optimally using the available area. For the data and constant memories a 512x16-bit RAM macro has been selected. This is more than required by a 256-bit ECDSA signature or verification algorithm. Whereas the size of the synthesized logic (CPU, bootloader, I/O) can be assumed as constant, the size of the program RAM has been maximized to store 5120x16 bits.

Table 1 lists the used components of the design and their relative area requirements. The largest part of the design are the RAMs. They cover more than 85% of the chip area. The CPU uses with its 6089 GE less than 7 % of the design area.

	Area	Area	
	[μm^2]	[GE]	
Program RAM	559173	59649	67.70%
Constant RAM	74168	7912	8.98%
Data RAM	74168	7912	8.98%
CPU	57085	6089	6.91%
ALU	30876	3294	3.74%
Bootloader	13130	1401	1.59%
EIA 232	12740	1359	1.54%
Timer 0	8153	870	0.99%
Timer 1	8153	870	0.99%
Timer 2	8128	867	0.98%
Instruction Decoder	3519	375	0.43%
Parallel I/O	2840	303	0.34%
Total	825975	88110	100.00%

Table 1. Area requirements of the Neptun ECC Processor.

The chip Neptun runs on a maximum frequency of 55 MHz. Using the tool Encounter with a VCD-file as input, a power simulation has been performed. During the elliptic curve point multiplication, where the most parts of the processor are active, the power consumption is 31.5 mW. Most of this relative high power requirement can be traced back to the program RAM. A power rail analysis showed that the ground bounce during the point multiplication is at most 3.55 mV. This low value has been achieved by placing a grid of power rails on top of the program RAM.

The important question at his point is: How does our processor perform in comparison to well established processor platforms. As it is shown in Table 2, we found two publications concerning NIST P-192 ECDSA implementations. The implementation by Gura [5] use an AVR processor with an 8-bit multiplier, 32 general purpose registers, but no accumulator. The Trimedia processor used by Hu [7] is a 32-bit processor that comes with a powerful VLIW architecture. In terms of cycles both implementations perform worse than our implementation. Because our previously introduced area-optimized design [13] has no constraints concerning the instruction set, its runtime is about 2% less than the runtime on Neptun.

5 Conclusion

In this paper we have presented the design and the implementation of a low-resource microprocessor that has an optimized instruction set for calculating computational intensive crypto operations like the Elliptic Curve Digi-

	Processor	Word size [bit]	Signature [kCycles]	Verification [kCycles]
Gura 2004 [5]	ATmega128	8	9920	
Hu 2004 [7]	TM1300	32	2955	
Wenger 2010 [13]		16	1391	2613
This work	**Neptun**	**16**	**1405**	**2675**

Table 2. Comparison of the performance of different processors.

tal Signature Algorithm (ECDSA). The core of the 16-bit processor in Harvard architecture is a multiply-accumulate unit that allows the efficient computation of the required finite-field operations. A further reason why our processor outperforms published results is that the memory access to the RAM macro has been optimized. The RAM access is possible in one clock cycle. A feature of parallel execution of instructions has been introduced. The presented approach is not only very promising for contactless devices like smart cards but allows extending nearly every product which requires an embedded microprocessor for security functionality. The self-written instruction-set simulator and assembler allow a very productive application of our processor in nearly every application domain. Currently, the processor is under fabrication on the UMC-L180 technology. Further tests and measurements will be conducted on the final silicon. Future work will consist of adapting an existing compiler for our processor and the evaluation of other crypto-algorithms and protocols.

Acknowledgement

We would like to thank the team working at the Microelectronics Design Center at the Swiss Federal Institute of Technology Zürich for their helpful comments concerning the chip design. This work has been supported by the Austrian Government through the research program FIT-IT Trust in IT Systems under the project number 825743 (project PIT).

References

[1] ARM Corporation. 16-bit Thumb Instruction Set. Available online at `http://infocenter.arm.com/help/topic/com.arm.doc.qrc0006e/QRC0006_UAL16.pdf`, May 2010.

[2] Atmel Corporation. 8-bit AVR Instruction Set. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, May 2008.

[3] Andreas Auer. Scaling hardware for electronic signatures to a minimum. Master's thesis, TU Graz, October 2008.

[4] Franz Fürbass and Johannes Wolkerstorfer. ECC Processor with Low Die Size for RFID Applications. In *Proceedings of 2007 IEEE International Symposium on Circuits and Systems*. IEEE, IEEE, May 2007.

[5] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, 6th International Workshop, Cambridge, MA, USA, August 11-13, 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2004.

[6] Daniel Hein, Johannes Wolkerstorfer, , and Norbert Felber. ECC is Ready for RFID Ű- A Proof in Silicon. In *Workshop on RFID Security 2008 (RFIDsec08)*, July 2008.

[7] Yu Hu, Qing Li, and C. C. Jay Kuo. Efficient implementation of elliptic curve cryptography (ecc) on vliw-micro-architecture media processor. In *ICME*, pages 879–882, 2004.

[8] International Organisation for Standardization (ISO). ISO/IEC 15693-3: Identification cards - Contactless integrated circuit(s) cards - Vicinity cards – Part 3: Anticollision and transmission protocol, 2001.

[9] International Organization for Standardization (ISO). ISO/IEC 14443: Identification Cards - Contactless Integrated Circuit(s) Cards - Proximity Cards, 2000.

[10] Sandeep S. Kumar and Christof Paar. Are standards compliant Elliptic Curve Cryptosystems feasible on RFID? In *Workshop on RFID Security 2006 (RFIDSec06), July 12-14, Graz, Austria*, 2006.

[11] Yong Ki Lee, Kazuo Sakiyama, Lejla Batina, and Ingrid Verbauwhede. Elliptic-Curve-Based Security Processor for RFID. *IEEE Transactions on Computers*, 57(11):1514–1527, November 2008.

[12] L. Leinweber, C. Papachristou, and F.G. Wolff. Efficient Architectures for Elliptic Curve Cryptography Processors for RFID. 2009.

[13] Erich Wenger, Martin Feldhofer, and Norbert Felber. Low-resource hardware design of an elliptic curve processor for contactless devices. In *11th International Workshop on Information Security Applications (WISA 2010), Jeju Island, Korea, August 24-26, 2010, Proceedings*. Springer, 2010.