A Lightweight ATmega-based Application-Specific Instruction-Set Processor for Elliptic Curve Cryptography

Erich Wenger

Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, 8010 Graz, Austria
`erich.wenger@iaik.tugraz.at`

Abstract. It is inevitable that future Radio-Frequency Identification (RFID) technology must support complex protocols and public-key cryptography. In this paper, we present an Application-Specific Instruction-Set Processor (ASIP) based on a clone of the ATmega128 microprocessor. A leakage-resilient, constant-runtime, and assembly-optimized software implementation of an elliptic curve point multiplication, which outperforms related work, requires 9,230–34,928 kCycles or 681–2,576 ms for standard conform elliptic curves (`secp160r1`, `secp192r1`, `secp224r1`, and `secp256r1`). Because this is too slow for most applications, the microprocessor has been equipped with a multiply-accumulate and a bit-serial instruction-set extension. Therefore, the runtime has been reduced to practically usable 96–248 ms, while keeping the power below 1.1 mW, and the area consumption between 19–27 kGE.

Keywords: ATmega, Elliptic Curve Cryptography, Instruction Set Extension, Application Specific Instruction-set Processor, Constant Runtime.

1 Introduction

The future Internet of things will consist of embedded smart cards, wireless sensor networks, and Radio-Frequency Identification (RFID) tags. Those devices must be capable to communicate with other entities over an air interface and must provide *privacy* and *security* capabilities. At Asiacrypt 2007, Serge Vaudenay [20] showed that "...an RFID scheme that achieves narrow-strong privacy ... essentially needs public-key cryptography techniques."

Among the three most popular public-key cryptographic systems (RSA, ElGamal, and ECC), Elliptic Curve Cryptography (ECC) is the least resource demanding and most suitable for embedded systems. In the past ECC has been well studied and standardized by SECG [2] and NIST [17]. One could also investigate non-standardized curves (e.g., by Gallant, Lambert, and Vanstone [7] or Bernstein *et al.* [1]), but for open-loop systems one should stick to the given standards. In this work we focus on the four prime-field based Weierstrass curves

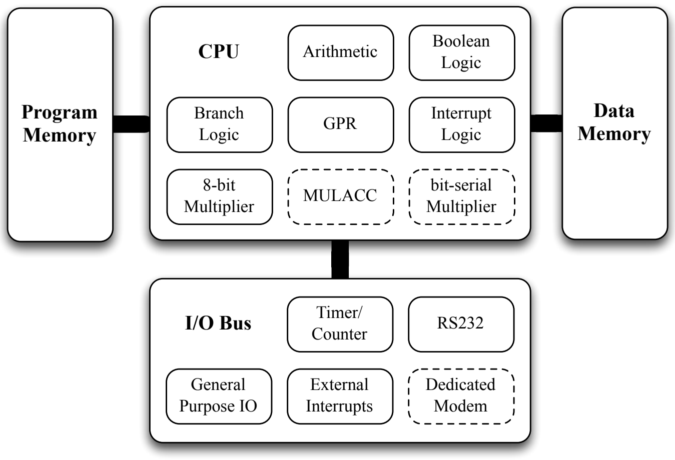

Fig. 1. Schematic diagram of the used processing architecture.

`secp160-256r1` as those are already used for mainstream applications such as TLS, IPSec, and SSH.

While public-key systems are very resource demanding, RFID tags must consume little power, be cheap (have a small chip area) and support real-time applications (respond within a given time). The traditional approach, which will pretty soon exceed its realms of possibility, is to equip the state machine of an RFID tag with a dedicated hardware block doing public-key cryptography. A more sophisticated solution is to base the design on a microprocessor, in particular on an Application-Specific Instruction-Set Processor (ASIP). An ASIP unites the advantages of programmable microprocessors (flexibility, extendability) with the advantages of dedicated hardware blocks (high-performance, low-power). Therefore, dedicated hardware units can be avoided and the overall hardware footprint decreases. In this paper, we transform a commercially available microprocessor into an ASIP targeting RFID and public-key cryptography.

For this paper, we base our design on the popular 8-bit Atmel ATmega128 AVR processor. This processor comes with an extensive instruction set and a dedicated hardware multiplier (important for prime-field arithmetic). The ATmega128 is used for a magnitude of applications and is supported by many toolchains (e.g., `avr-gcc`, IAR, Crossworks). In Wenger *et al.* [21], we presented 'Just Another AVR' (JAAVR, see Figure 1), a feature-complete clone of the popular ATmega128 which is written in VHDL and only requires 6,140 GE, making it perfectly suitable for area-sensitive embedded designs. In [21], we equipped an earlier version of JAAVR with a dedicated RFID modem, and evaluated ECC, AES, and Grøstl. As expected, those results show that ECC is the dominating component.

Our Contribution. In this paper we present an JAAVR-based ASIP optimized for ECC. First, we present new assembly optimized ATmega-compatible runtime results in which we outperform related software implementations (including our own in [21]). Second, we optimize the cycles-per-instruction (CPI) of all load, store, and multiply instructions of JAAVR in order to achieve speedups of 25–27 %. Third, we are the first to actually build an ATmega128-compatible processor with multiply-accumulate instruction-set extensions as ASIC. Previous work was either simulated or only performed on FPGAs. Fourth, we are also the first to build a tightly-coupled bit-serial multiplier as instruction-set extension of JAAVR for prime-field based ECC. Utilizing all those techniques, we present a 19 kGE small design suitable for RFID and other real-time applications.

This paper is structured as follows: Section 2 elaborates some basic design decisions. Section 3 discusses efficient software implementation techniques for ECC. Sections 4-6 deal with the improvement of the CPI of JAAVR, the utilization of a multiply-accumulate instruction, and the integration of a bit-serial multiplier, respectively. Section 7 discusses the results in connection with related work. Section 8 concludes the work. The most important results are gathered in Table 2. They are discussed throughout the paper.

2 Basic Reasoning

For RFID applications, the runtime of an algorithm is important in two respects. First, it must be sufficiently fast to support real-time applications. Second, by having a fast implementation, one can reduce the clock frequency and therefore reduce the power consumption. For a passively powered RFID tag, the power consumption is of upmost importance. For a typical ISO-14443-compatible [13] tag, we assume the following requirements. The clock should be an integer fraction of 13.56 MHz, the maximum power consumption below 2 mW, and the maximum runtime for an ECC point multiplication is 100-500 ms. It should be noted that all our hardware designs easily exceed this minimum clock frequency of 13.56 MHz.

Tools. For all of our implementations, we performed hardware synthesis (Cadence RTL Compiler v08.10), power simulations (Cadence First Encounter v08.10), and cycle-accurate post-synthesis and post-layout hardware simulations (using NCSim v08.20). As process technology the UMC 130 nm low-leakage CMOS technology with Faraday design libraries in combination with area-efficient single-port register-based RAM macros and Via-1 ROM macros is used. Previous experiments showed that synthesizing the program memory as standard logic cells results in smaller (post synthesis) but less routable designs (post place-and-route). A decreased cell density increases the size of the synthesized program memory to a point where the available Via-1 ROM macro is effectively smaller.

Practical Security. When implementing cryptography, the designer must consider practical attack scenarios such as timing, side-channel and fault attacks. Regarding timing attacks, all assembly-optimized implementations perform the point multiplication in constant runtime. Further, all implementations provide

Table 1. `secp160r1` point multiplication results using different multi-precision integer multiplication methods: operand-scanning (OS), product-scanning (PS), hybrid, and operand-caching (OC).

Impl.	**Point-**	**Integer-**	**Program-Memory**		**Chip**
	Multiplication	**Multiplication**	**Size**	**Integer Mul.**	**Area**
	[kCycles]	[Cycles]	[Bytes]	[%]	[GE]
OS in C	37,168	9,807	4,188	3	17,738
OS	17,607	5,505	12,110	62	23,540
PS looped	17,226	5,367	4,636	4	17,638
PS	13,546	4,035	9,860	54	21,701[a]
Hybrid	10,609	2,972	9,050	49	21,701[a]
OC	9,230	2,473	8,218	46	21,701[a]

[a] Identical, because only certain discrete ROM macros are available.

a basic resistance against power-analysis attacks. The Montgomery ladder formula by Hutter *et al.* [11] performs key-independent double-and-add operations. With its requirement of 16 field multiplications and 17 field additions per key bit, it is reasonably fast. The finite-field multiplication is used for multiplications as well as for squarings. At the end of the Montgomery ladder, a y-coordinate-recovery and a constant-runtime inversion based on exponentiation (Fermat's little theorem) are performed. For side channel and fault security we also perform projective point randomization [4] before (against side-channel attacks) and point verification before and after (against fault attacks) the point multiplication. Because we did not perform practical power analysis attacks or fault simulations, we do not claim to be side channel or fault secure, but we use algorithms that improve resistance against those attacks. Thus all our results are practically relevant.

3 The Baseline: Efficient Software Implementation

By choosing an 8-bit processor we start with a rather small but "arithmetically speaking" slow processor. Our first not constant-time, plain-C implementation showed excruciatingly-slow runtimes of 37–131 million cycles, 2.7–9.6 seconds (@ 13.56 MHz). Thus optimizing the existing code in assembly is mandatory. For all following comparisons we consider our C implementations as baseline. In hardware it requires 16.9–19.5 kGE and 561–656 μW.

The first (and most laborious) optimization we have performed is the replacement of all field operations with constant-runtime assembly functions. This not only improves the runtime but also makes all timing attacks infeasible. The field addition and subtraction operations have been unrolled and perform the reduction without branches. For the field multiplications, we have taken advantage of the standardized Mersenne-like primes to get branch-free code using only addition and shift operations.

The most time-consuming algorithm is the multi-precision integer multiplication. Hutter and Wenger [12] did a thorough comparison between the Schoolbook's operand-scanning (OS), Comba's [3] product-scanning (PS), Gura *et al.*'s [9] hybrid and their own operand-caching (OC) multiplication methods. We implemented unrolled and looped versions of those algorithms in assembly. Table 1 shows that by doing so the runtimes of integer and point multiplications for `secp160r1` were improved by factors of 3.97 and 4.03, respectively. Our fastest implementation, based on the operand-caching method, achieved a runtime of 9,230 kCycles for a point multiplication. For comparison: Gura *et al.* [9], Szczechowiak *et al.* [19], Wenger *et al.* [21], and Liu *et al.* [16] achieved runtimes of 6,480 kCycles, 9,376 kCycles, 13,027 kCycles and 16,939 kCycles, respectively. However, most of those implementations do not consider side-channel attacks. For instance, Gura *et al.* used a Jacobian-based NAF point-multiplication formula. For reference, we applied the same technique as Gura *et al.* and improved their fastest implementation by 50 kCycles to 6,430 kCycles.

Apart from the expected runtime differences (OS > PS > Hybrid > OC), unrolling the integer multiplication has a huge impact on the size of the program code. Up to 62 % of the entries in the program memory are due to the unrolled integer multiplication. Compared to the C implementation, the chip size of the program memory increased by up to 76 %. Despite of that, assembly optimization and 'unrolling' improved the area-time-product by a factor of up to 3.3, thus establishing themselves as one of the most important optimization techniques.

The focus of this section was to perform software optimizations both applicable to the ATmega128 and JAAVR. In the next sections, we present hardware optimization techniques that improve both the execution time as well as the total hardware footprint.

4 Improving the CPU

Already during the design of JAAVR, we realized several avenues for optimization potentials. It was necessary to artificially introduce `NOP` operations in order to achieve identical cycles-per-instruction (CPI) counts compared to the original ATmega128. The most significant difference is that the ATmega128 uses a two-stage pipeline and JAAVR does not. So all we needed to improve the performance of *store* and *multiply* operations was to deactivate the `NOP` operations.

Unlike our previous paper [21], we also optimized memory *load* operations. For the cycle-accurate (CA) design, two cycles are needed to load data from the synchronous data memory. During the first cycle the address is applied to memory and during the second cycle the obtained data word is stored to a general purpose registers (GPR).

In order to reduce the latency of all *load* operations, we decided to introduce a pipelining structure. While the first cycle of the operation stays identical, the second cycle is performed as part of the subsequent operation. As Figure 2 shows, multiplexers before and after each general purpose register were added. MUX2 is used to update the next value stored within the GPR. MUX1 overrides the

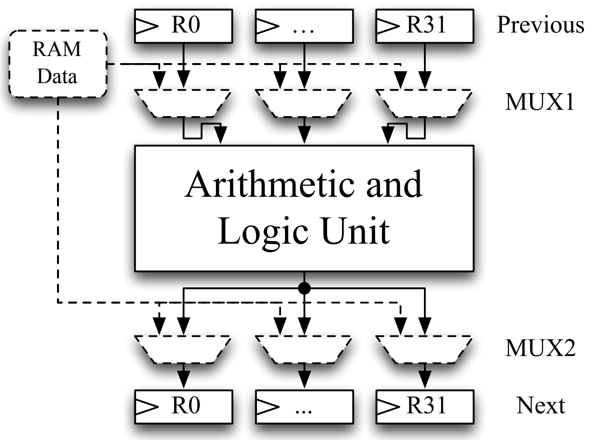

Fig. 2. The multiplexers were added to reduce the necessary cycles per load instruction.

current contents within the GPR. Thus the ALU is working on an updated set of GPR. The impact of the multiplexers on the critical path is hardly noticeable.

By switching JAAVR from the CA to the FAST mode, the following instructions improve: `MUL*`, `ST`, `STD`, `PUSH`, `LD`, `LDD`, `POP`, `IJMP`, `RJMP`, `CBI`, `SBI` ($2 \rightarrow 1$), `RCALL`, `ICALL`, `LPM`, `ELPM` ($3 \rightarrow 2$), `CALL`, `RET`, and `RETI` ($4 \rightarrow 3$). This increased the size of JAAVR form 6,140 GE to 6,791 GE (by 10 %), while the runtime of the fastest point multiplication improved by 26 %. Thus, the area-runtime product improved by a factor of 1.31.

After enabling those optimizations, JAAVR is still instruction-set compatible with the original ATmega128. So any (cryptographic) algorithm would benefit from the improved instruction-timing. The next two sections are dedicated on optimizing JAAVR for ECC using instruction-set extensions, transforming our design into an ASIP.

5 The Power of the MULACC Command

When investigating the instructions used for the unrolled product-scanning multi-precision multiplication, one can observe that there are four instructions, always used in consecutive order: `MUL`, `ADD`, `ADC`, and `ADC`. The idea behind the multiply-accumulate instruction-set extension is to combine those instructions into a single `MULACC` command, as it has been done in related work.

Already in 2004, Großschädl and Savaş [8] used five custom instructions to accelerate prime fields and binary extension fields on a MIPS32 core and gained a speedup of about six for binary extension fields. In 2005, Eberle *et al.* [5] presented multiply-accumulate instruction-set extensions for binary-extension fields on an ATmega128. They improved `sect223r1` by a factor of 13.6, but did not use ISE for prime fields as we do it in this paper.

In fact, we used the `MULACC` instruction to improve the fastest multi-precision multiplication formula: the operand-caching method. We are the first to combine the operand caching method with an instruction-set extension. Like the product-scanning method, this method uses the same sequence of instructions as mentioned above. So, by combining the operand-caching multiplication, which reduces the number of load and store operations, and the multiply-accumulate instruction, which reduces the number of additions, we achieved a new speed record: 631 cycles for a 160-bit integer multiplication.

There are two main challenges concerning the introduction of new instructions: First, most of the 2^{16} possibilities of the 16-bit instruction words are already assigned to existing instructions. Thus, the introduction of a new instruction would mean to modify existing instructions and being no longer compatible with the original ATmega128. Second, adapting the source code of `avr-gcc`, `avr-as`, and `avr-ld` to add new instructions does not seem to be straightforward.

Our solution is to introduce a new, within the I/O memory mapped, register that can switch the processor to a special operating mode. In this special operating mode, certain existing instructions are reinterpreted. For this solution, none of the `avr-gcc` tools had to be modified.

In order to improve the performance of the operand-caching multiplication, we introduced two instructions: `MULACC` and `ST_SHIFTACC`. `MULACC` multiplies two registers Rd and Rr and adds the result to the accumulator stored in R0-R2: $(R2, R1, R0) \leftarrow (R2, R1, R0) + Rd \times Rr$. This operation can be performed 2^8 times without the risk of an overflowing accumulator. This is more than the required $e = 10$ accumulations performed within the operand-caching multiplication algorithm (e is a parameter to adjust the operand caching method, see [12]). After e `MULACC` operations, `ST_SHIFTACC` is used to store the lowest byte of the accumulator (`ST R0,Z+`) and shifts the accumulator by 8 bits to the right ($R0 \leftarrow R1$, $R1 \leftarrow R2$, $R2 \leftarrow 0$). Because of those optimizations, we freed up two registers that were used as temporary storage of the product. In order not to waste them, we increased e from 10 to 11, which further decreased the number of necessary load and store instructions.

By using those instruction-set extensions, a 160×160-bit multiplication can be performed three times faster. It takes 631 cycles compared to 1,896 cycles. A detailed decomposition of the used instructions can be found in Appendix A. Further, the ISE had hardly any impact on the size of JAAVR. Only 257 GE or 3.8 % of additional logic had to be added. At the same time, the size of the program memory decreased: from 11,807 GE to 8,202 GE (-31 %). Adding all those improvements together, the area-time product improved by a factor of 2.4.

A point multiplication in `secp160r1` takes 3,268 kCycles. A profiling analysis showed that 83.2 % of the total runtime are spent on the field multiplications. The optimized reduction algorithm for the `secp160r1` prime $2^{160} - 2^{31} - 1$ utilizes 26.1 % of the total runtime or a third of the field multiplication. Thus, any further optimizations must not only consider the integer multiplication, but the field multiplication as a whole. This is done in the following section.

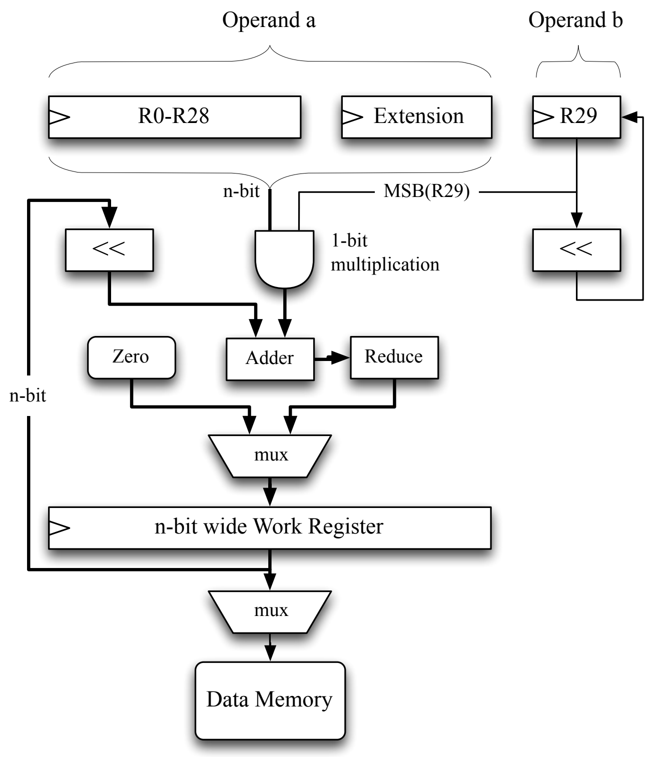

Fig. 3. The bit-serial multiplier is merged with the CPU.

6 Using a Dedicated digit-serial Multiplier

When investigating related work on ECC, one can either find ECC designs based on an word-level multiplier (cf. [10, 22]), as we used in the previous sections, or designs based on a digit-serial multiplier (cf. [6]). A digit-serial multiplier simultaneously operates on all digits of the multiplicand a, but only a single digit b_i of the multiplicand b.

In this section we want to introduce the concept of a 'tightly-coupled' bit-serial multiplier, which merges a bit-serial multiplier with the CPU. By reusing existing registers, we were able to keep the impact on the total chip area to a minimum and avoid unnecessary data transfers.

A block diagram of our bit-serial multiplier is depicted in Figure 3. Algorithm 1 shows the pseudo-code to control it. An MSB-first multiplier is used which accesses all b_i starting with the most significant bit. The `Z`-register (`R30`, `R31`) is used to address the memory, and `R29` is used to store the byte containing b_i. During each cycle, `R29` is shifted to the left using the `LSL` (Logic Shift Left) instruction of JAAVR. At the same time the `Work` register is updated in the

Algorithm 1 Pseudocode for the bit-serial multiplication.

```
 1: PUSH all call-saved registers.
 2: LD operand $a$ to R0-R28 and Extension.
 3: Switch to ISE mode.                          (memory mapped config register)
 4: Clear Work register.
 5: for $i$ from $\lceil \frac{n}{8} \rceil - 1$ to 0 do
 6:     LD R29, -Z                  (load $b_i$, pre-decrement pointer register Z)
 7:     8 times: LSL R29                        (triggers bit-serial multiplier)
 8: end for
 9: Store Work register.
10: Switch back to normal mode.
11: POP all call-saved registers.
```

following manner: `Work` $\leftarrow (a \times b_i + \texttt{Work} \ll |b_i|) \pmod{p}$. In each cycle the most significant bit of `R29` (b_i) is multiplied with a, the product is added to a shifted version of the n-bit[1] `Work` register, and the `Work` register is updated with the reduced sum. After the last computation cycle, the product $a \times b \pmod{p}$ is stored in the `Work` register. A modified store instruction (`ST`) is used to write the `Work` register to memory, one byte at a time.

To reuse existing registers, a is stored in register `R0` to `R28` and the `Extension` register. For `secp256r1`, three IO-memory-mapped 8-bit `Extension` registers are necessary. So for `secp160-224r1` it was only necessary to add the `Work` register and the combinatoric logic.

A field multiplication for `secp160r1` takes 271 cycles. $9 \times 20 = 180$ cycles are used by the digit-serial multiplication, $2 \times 20 = 40$ cycles are necessary for loading operand a and storing the result, and 51 cycles are necessary to comply with the C-calling convention (`PUSH`, `POP`, `CALL`, `RET`) and to switch between the normal ATmega128 compatible operation mode and the instruction-set-extension mode.

Compared to the original software implementations in C, a speedup between 30 (`secp160r1`) and 44 (`secp256r1`) was achieved. The bit-serial approach is also 2.3–3.7 times faster than the `MULACC` instruction-set extension. The size of the program memory decreased significantly by 25 %–41 %. However, the size of the CPU core increased by 61 %–107 %. 4,551 GE are required for the bit-serial multiplier for `secp160r1` and 7,792 GE have to be added for `secp256r1`. The question now is whether adding the bit-serial multiplier improves or worsens the area-runtime product. In fact, it improves by a factor of 2.1–3.1, which makes the tightly-coupled digit-serial multiplier (by far) the fastest, even though not the smallest hardware implementation presented in this paper.

7 Results

The most important results of our implementations are summarized in Table 2 and have already been discussed in the previous sections. It contains figures that

[1] n relates to the number of bits needed to represent any value in $\mathbb{F}_p$.

Table 2. Summary of all experiments. SARP stands for ‘scaled area-runtime product’.

Impl.	Runtime	Program Memory	Data	Area Requirement JAAVR	ROM	RAM	Total	Power @13.56 MHz	Energy	Runtime @13.56 MHz	SARP
	[kCycles]	[Bytes]	[Bytes]	[kGE]	[kGE]	[kGE]	[kGE]	[μW]	[μJ]	[ms]	
`secp160r1`											
CA in C	37,168	4,188	418	6,140	7,744	3,855	17,738	561	1,539	2,741	22.5
CA	9,230	8,218	402	6,140	11,807	3,754	21,701	662	450	681	6.8
FAST	6,764	8,218	402	6,791	11,807	3,754	22,352	824	411	499	5.2
MULACC	3,268	5,688	402	7,048	8,202	3,754	19,004	850	205	241	2.1
bit-serial	1,298	4,286	350	11,341	7,744	3,452	22,537	1,013	97	96	1.0
`secp192r1`, **NIST P-192**											
CA in C	55,365	3,916	483	6,140	6,505	4,233	16,877	640	2,615	4,083	21.6
CA	15,093	10,070	462	6,140	11,807	4,107	22,054	677	753	1,113	7.7
FAST	11,101	10,070	462	6,791	11,807	4,107	22,705	832	681	819	5.8
MULACC	5,022	6,396	462	7,048	10,040	4,107	21,195	864	320	370	2.5
bit-serial	1,813	4,490	406	12,302	7,744	3,779	23,825	1,084	145	134	1.0
`secp224r1`, **NIST P-224**											
CA in C	86,058	3,926	569	6,140	6,363	4,712	17,215	663	4,208	6,346	23.9
CA	23,213	12,374	526	6,140	15,484	4,485	26,109	689	1,179	1,712	9.8
FAST	17,114	12,374	526	6,791	15,484	4,485	26,760	843	1,063	1,262	7.4
MULACC	7,537	7,404	526	7,048	10,040	4,485	21,573	848	472	556	2.6
bit-serial	2,469	4,808	466	13,237	7,744	4,132	25,113	1,032	188	182	1.0
`secp256r1`, **NIST P-256**											
CA in C	130,695	5,604	645	6,140	8,202	5,165	19,506	656	6,320	9,638	27.8
CA	34,928	15,888	590	6,140	17,029	4,838	28,006	663	1,707	2,576	10.7
FAST	26,290	15,888	590	6,791	17,029	4,838	28,657	811	1,572	1,939	8.2
MULACC	11,900	9,372	590	7,048	11,807	4,838	23,693	836	733	878	3.1
bit-serial	3,367	5,532	522	14,583	8,202	4,460	27,244	1,031	256	248	1.0

are characteristic for software and hardware implementations. Every row labeled with cycle accuracy (CA) is applicable for an ATmega128 as well as JAAVR. Using a TCL-based simulation script, we measured the data-memory requirements (including stack) of all implementations. The bit-serial designs needs the least data memory, because the memory for a temporary $2n$-bit product was saved. The RAM and ROM macros are chosen according to the data and program memory requirements. Because those macros are only available in certain sizes, not every difference measured in Bytes is reflected in the actual area of the ROM macro (in gate equivalents).

7.1 Reached Goals

All targeted goals (< 2 mW, < 500 ms @ 13.56 MHz) have been met. An exception are the larger 224-bit and 256-bit elliptic curves which render the `MULACC` based approach as too slow. The runtimes of 100–200 ms show that the clock frequency can be decreased by factors of 2–4, which in turn would decrease the power consumptions by a factor of 2–4.

7.2 Related Work

For a fair comparison with related work, it is important to not only consider plain numbers (chip area, runtime, power), but also the provided features. We distinguish whether a design is microprocessor-based (MCU), comes with a C-compiler, considers side-channel attacks, or performs binary- or prime-field based

Table 3. Comparison with related work.

Reference	**Runtime** [kCycles]	**Area** [GE]	**Characteristics**
	ISE - `secp160r1`		
Gura [9]	4,720	-	ATmega-based
OC + `MULACC`	3,268	19,004	ATmega-based
	Dedicated Hardware - `secp160r1`		
bit-serial	1,298	22,537	ATmega-based
OC + `MULACC`	3,268	19,004	ATmega-based
Fürbass [6]	362	19,000	ECDSA-like
	Dedicated Hardware - `secp192r1`		
Satoh [18]	4,165	29,655	ECC
bit-serial	1,813	23,825	ATmega-based
Fürbass [6]	502	23,600	ECDSA-like
OC + `MULACC`	5,022	21,195	ATmega-based
Hutter [10]	859	19,115	ECDSA, MCU
Wenger [22]	1,377	11,686	ECDSA, MCU

ECC. One must also consider the used manufacturing technology, but this would go beyond the scope of this paper.

A fair comparison with dedicated hardware designs is tough. While they are optimized to the limit, they lack the rich set of features our ASIP provides. The ASIP is easily extendable and provides a compiler toolchain. Also our microprocessor-based approach has not yet reached its limits (c.f. Appendix C), but applying those ideas would make our design incompatible with a standard ATmega128. Table 3 summarizes the comparison with related work.

Fürbass *et al.* [6], Hutter *et al.* [10], Satoh *et al.* [18], and Wenger *et al.* [22] worked on dedicated prime-field based elliptic curve hardware designs. They require 12–30 kGE of hardware and 362–4,165 kCycles of runtime. Although most of their solutions are faster, it is important to notice that our solutions provide sufficiently fast response times. Hutter *et al.* and Wenger *et al.* implemented the full ECDSA signature algorithm and Fürbass *et al.* implemented ECDSA without a hash algorithm. The designs by Hutter *et al.* and Wenger *et al.* is micro controller based, but does not provide a C-compiler. The designs by Fürbass *et al.* and Satoh *et al.* are not micro controller based, so it probably is easier to adapt our designs for real-world scenarios.

Most comparable to this paper are the works of Gura *et al.* [9], Kumar and Paar [15], and Koschuch *et al.* [14]. Gura *et al.* added simulated ISE to an AVR processor, but achieved slower runtimes results. Kumar and Paar used the ATSTK94 FPSLIC demonstration board to extend an AVR processor with a bit-serial multiplier extension for binary extension fields. They however have not applied their methodology to prime fields and do not provide results for an ASIC. Koschuch *et al.* synthesized an 8051-compatible microprocessor and equipped it with a hardware accelerator for binary extension fields. In total, they needed 29 kGE and 1.2 MCycles. Even though we use prime fields, our results are smaller and approximately of similar speed.

8 Conclusion

After our thorough analysis of instruction-set extensions for ECC, the following conclusions can be drawn: First, if the area footprint is most important (e.g., for RFID) our `MULACC`-based ASIP is the best choice. The chip area is on par with related work and reasonable response times of less than 370 ms are achievable. Second, for applications such as wireless sensor networks or embedded smart cards, the tightly-coupled bit-serial ASIP approach is most suitable. It provides the best energy efficiency and the best area-time product. Third, the design space for ECC implementations is huge: the ratios between the best and the worst implementation across all tested elliptic curves in the categories of area-runtime product, runtime, and energy are 87:1, 101:1, and 65:1, respectively. Our results show that the figures vary by up to two orders of magnitude across the hardware/software design space, which gives a designer a multitude of options to fine-tune a design for a given set of requirements.

Acknowledgements

The author wants to thank Thomas Plos for the support during the creation of this paper. This work has been supported by the Austrian Science Fund (FWF) under grant number TRP 251-N23 (Realizing a Secure Internet of Things - ReSIT).

References

1. D. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards Curves. In *AFRICACRYPT*, volume 5023 of *LNCS*, pages 389–405, 2008.
2. Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0, 2000.
3. P. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, October 1990.
4. J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Ç. K. Koç and C. Paar, editors, *CHES*, volume 1717 of *LNCS*, pages 292–302. Springer, 1999.
5. H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, and V. Gupta. Architectural Extensions for Elliptic Curve Cryptography over GF(2^m) on 8-bit Microprocessors. In *International Conference on Application-specific Systems, Architectures and Processors*, pages 343–349. IEEE Computer Society, July 2005.
6. F. Fürbass and J. 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.

7. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In *CRYPTO*, LNCS, pages 190–200, 2001.
8. J. Großschädl and E. Savaş. Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m). In *CHES*, pages 133–147, 2004.
9. N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs. In M. Joye and J.-J. Quisquater, editors, *CHES*, volume 3156 of *LNCS*, pages 119–132. Springer, 2004.
10. M. Hutter, M. Feldhofer, and T. Plos. An ECDSA Processor for RFID Authentication. In *RFIDSec*, pages 189–202, 2010.
11. M. Hutter, M. Joye, and Y. Sierra. Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-Z Coordinate Representation. In *AFRICACRYPT*, volume 6737 of *LNCS*, pages 170–187, 2011.
12. M. Hutter and E. Wenger. Fast Multi-Precision Multiplication for Public-Key Cryptography on Embedded Microprocessors. In B. P. und Tsuyoshi Takagi, editor, *CHES*, volume 6917 of *LNCS*, pages 459–474. Springer, 2011.
13. International Organization for Standardization (ISO). ISO/IEC 14443-3: Identification Cards - Contactless Integrated Circuit(s) Cards - Proximity Cards - Part3: Initialization and Anticollision, 2001.
14. M. Koschuch, J. Lechner, A. Weitzer, J. Großschädl, A. Szekely, S. Tillich, and J. Wolkerstorfer. Hardware/Software Co-design of Elliptic Curve Cryptography on an 8051 Microcontroller. In *CHES*, 2006.
15. S. Kumar and C. Paar. Reconfigurable Instruction Set Extension for Enabling ECC on an 8-Bit Processor. In *Field Programmable Logic and Application*, volume 3203 of *LNCS*, pages 586–595, 2004.
16. A. Liu and P. Ning. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In *International Conference on Information Processing in Sensor Networks*, pages 245–256, 2008.
17. National Institute of Standards and Technology (NIST). FIPS-186-3: Digital Signature Standard (DSS), 2009.
18. A. Satoh and K. Takano. A Scalable Dual-Field Elliptic Curve Cryptographic Processor. *IEEE Transactions on Computers*, 52:449–460, 2003.
19. P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab. NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In *Wireless Sensor Networks*, LNCS, pages 305–320. Springer, 2008.
20. S. Vaudenay. On Privacy Models for RFID. In *ASIACRYPT*, LNCS, pages 68–87, 2007.
21. E. Wenger, T. Baier, and J. Feichtner. JAAVR: Introducing the Next Generation of Security-Enabled RFID Tags. In *DSD*, pages 640–647, 2012.
22. E. Wenger, M. Feldhofer, and N. Felber. Low-Resource Hardware Design of an Elliptic Curve Processor for Contactless Devices. In *WISA*, pages 92–106, 2010.

A Decomposition of Instructions

Table 4 shows the decomposition of used instructions for performing a 160×160-bit multiplication with and without instruction-set extension. Using the ISE, the necessary additions were nearly eliminated.

Table 4. Decomposition of the number of cycles per type of instruction necessary for a 160×160-bit multiplication.

Instruction	**CA**	**FAST**	**ISE**
`MUL`	800	400	0
`MULACC`	0	0	400
`LD`	160	80	76
`ST`	120	60	0
`ST_SHIFTACC`	0	0	58
`ADC`	820	820	18
`ADD`	420	420	18
`CLR`	63	63	4
`PUSH`	36	18	18
`POP`	36	18	18
Others	18	17	21
Total	**2,473**	**1,896**	**631**

B Runtimes of Finite-Field Operations

The runtimes of all finite-field operations for `secp160r1` are presented in Table 5. Especially the comparable slow finite-field multiplication greatly profited from the performed optimizations. Because the inversion is based on an exponentiation, the speedup of the inversion is a direct reflection of the speedup of the multiplication. Using the bit-serial multiplier, the ratio between additions and multiplications is only 1.5. This needs to be taken in concern when a method or formula for the point multiplication is chosen.

Table 5. Runtimes of finite-field operations for `secp160r1`.

Operation	**CA**	**FAST**	**ISE**	**bit-serial**
Addition	291	176	176	176
Subtraction	291	176	176	176
Multiplication	3,024	2,249	984	271
Inversion	519,217	386,368	170,053	48,130

C The Limits

In this paper we concentrated on delivering sufficiently fast and power-aware ASIPs for future RFID technology. We sticked to modifying the processing core and only optimized the finite-field operations in assembly language. However, if you want to make our design into an actual product, further optimizations need to be considered.

Constants consume space within the program and data memory. At startup they are loaded from the program memory and stored to the data memory. If one would add a memory-mapped table within the data memory bus, one could significantly reduce the size of the necessary RAM macro. The RAM macro could be shrunken by at least seven times 160-bit = 140 bytes.

Memory Management is currently performed by the compiler by reserving memory on the stack. If the whole source code would be written in assembly, unnecessary and redundant data memory entries could be avoided.

Processor Features that are not needed for ECC could be removed. E.g. the I/O bus is mapped within the data memory, or `MOVW` instructions are not needed for the finite-field operations. Removing those feature would decrease the size of JAAVR by 420 GE.

Processor Instructions that are not needed for ECC could be removed. For instance the `FMUL*` and `MULS*` multiplier instructions are not needed for an ECC point multiplication.

Program Memory is currently synthesized as Via-1 ROM macros. Using smaller ROM macros would significantly decrease the size of the program memory. Because the program memory is the largest part of the presented design, decreasing its size has a significant impact on the total chip area.