Comparison of Low-Power Public Key Cryptography on MICAz 8-Bit Micro Controller

Leif Uhsadel

April 4, 2007

Diploma Thesis
Ruhr-University Bochum

Faculty of Electrical Engineering
and Information Technology
Chair for Communication Security
Prof. Dr.-Ing. Christof Paar
Dipl.-Ing. Axel Poschmann

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Bochum, April 04, 2007 ____________________
Unterschrift

Acknowledgment

I would like to thank all the people who contributed in whatever manner to the success of this work. Thank you Prof. Dr.-Ing. Christof Paar for the chance to write my Diploma thesis at the Communication Security research Group. Thank you Axel Poschmann for supervising me, and all the advise you gave me, and all the time you had for my various questions. Thank you Bernd Jercyna for correcting my typos. Thank you Tim Güneysu for supporting me with the creation of the maple scripts that I needed to check my results. Thank you Bodo Möller for answering my questions concerning algorithms and Latex. Thank you An Liu for answering my questions concerning TinyECC. Thank you Jörg Wunsch and the whole AVR community for helping me with the AVR tool chain. Thank you Gordon Meiser for accompanying me during the night shifts. Thank you Sören Rinne for listening to all my technical problems. Thank you Anna Joachim for easing this stressed time so much. Thank you to my whole family for all the support you gave me. And thank you to all of you I forgot to mention. You all have been a big help!

Kurzfassung

Die Begriffe *ubiquitous* und *pervasive computing* bedeuten allgegenwärtige und durchdringende elektronische Datenverarbeitung. Dabei geht es um die Integration und Vernetzung von kleinen, intelligenten Geräten in unser alltägliches Leben um Komfort und Sicherheit zu erhöhen. Die Anwendungsfelder sind weit gefächert und erfassen unter anderem Militär, Medizin und Landwirtschaft. Die genannten Bereiche sind besonders interessant, weil sie häufig sensible Daten verarbeiten. Sicherheit spielt hier ein große Rolle bei der Übermittlung von Daten, weil die Kommunikation häufig kabellos stattfindet. Ein potentieller Angreifer könnte unverschlüsselte Kommunikation nach belieben mithören und verändern. Zum Einsatz kommen hier bevorzugt symmetrische Verfahren, da sie verglichen mit asymmetrischen verfahren immense Vorteile bezüglich der Geschwindigkeit haben. Die geringere CPU Belastung wirkt sich positiv auf den Energieverbrauch aus und verlängert die Lebensdauer der Nodes. Jedoch haben symmetrische Verfahren signifikante Nachteile bezüglich der Schlüsselverteilung. Der Einsatz asymmtrischer Verfahren wird momentan kontrovers diskutiert. Den Vorteil bei der Schlüsselverteilung und Verwaltung stehen Faktor 1000 höhere Berechnungszeiten und damit um den Faktor 1000 höhere Energie Kosten gegenüber.

In dieser Arbeit werden zunächst existierende Implementierungen von asymmterischen Verfahren vorgestellt und untersucht. Elliptische Kurven heben sich aufgrund ihrer hohen Geschwindigkeit und kurzen Schlüssellänge dabei als klare Favouriten hervor. Weiterhin gelten Elliptische Kurven als sicher, weil diese Technik seit ca. 1986 bekannt ist und auf dem diskreten logarithmus Problem (DLP) basierende Protokolle wie ECDSA nicht gebrochen sind.

Eine anschließende Untersuchung ergibt, daß es sinnvoll ist, Kryptografie mit Elliptischen Kurven in die drei Teile Protokoll, Kurve und Primköprer einzuteilen. Optimierungen und Auswahl von Algorithmen auf allen drei Ebenen sind maßgeblich für die Geschwindigkeit. Dabei kann ein effizient implementierter Primkörper für mehrere Protokolle und Kurven verwendet werden im Gegensatz zu einer effizient implementierten Kurve, deren Optimierungen wenigstens Teilweise für jedes Protokoll angepasst werden müssen.

Im anschließenden Hauptteil der Arbeit wird für eine standardisierte Kurve ein Primkörper in Assembler implementiert, der als Grundlage für zukünftige Implementierungen von verschiedenen Protokollen dienen soll. Eine sorgfältige Wahl von Algorithmen und Optimierungen, sowie eine ausgeklügelte Taktik zum Handling von Carrybits führt, nach bestem Wissen, zur bisher schnellsten bekannten Implementierung des 160-bit Primkörpers secp160r1, standardisiert nach SECG2, für den Mikrocontroller ATMega128L. Die Implementierung ist quelloffen und steht auf Anfrage zur Verfügung. Eine Elliptische Kurve ist in C implementiert um die Funktionalität des Primkörpers zu zeigen. Die Kurve enthält einige wesentliche Optimierungen und dient nur als Startpunkt für eine sehr effiziente Implementierung.

Abschliessend wird der implementierte Primkörper und die implemetierte Kurve mit den bereits vorgestellenten Implementierungen verglichen. Die Implementierung der 160-bit Multiplikation benötigt $5,4$ KB Flash-Speicher und 112 B SRAM und ist ca. 7% schneller als die schnellste bisher bekannte und bietet daher auch eine 7%ige Energieersparnis. Der Primkörper kann ohne erneutes optimieren in belibige Implementierungen der zugehörigen Kurve integriert werden.

Abstract

The terms *ubiquitous* and *pervasive computing* designate the penetration of our everyday life with intelligent devices. These tiny, constrained, and battery powered nodes are used to build WSNs that may process sensitive data. Therefore security as well as low energy consumption are crucial in this field. Since runtime scales with energy consumption efficient implementation is necessary at all costs. We will show by comparing of different implementations of asymmetric algorithms that ECC is a good choice in this case, as it allows shorter key length with adequate security level and furthermore can be efficiently implemented. We will provide mathematical background as well as algorithms for an efficient implementation. Subsequently we will present the fastest known implementation of a 160-bit multiplication, which is the core operation of the prime field of the standardized elliptic curve secp160r1. Even though the implementation is highly optimized for speed, the code-size of 5.4 KB and RAM requirements of 112 B are acceptable. The high efficient prime field is implemented in assembly and available on request. It is thought to be the base for high efficient curve implementations. A curve with basic optimizations is written in C and can also be reused. The 160-bit multiplication has a runtime of 0.39ms and requires with our C implementation of the curve 1.151s for a point multiplication. This could be optimized to approximately 0.76s for one point multiplication in combination with a highly efficient elliptic curve. Furthermore this would allow the execution of an ECDSA signature in less than one second without pre-calculation.

Contents

Eidesstattliche Erklärung i

Danksagung iii

List of Figures xi

List of Tables xiii

List of Algorithms xv

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Aim of this Thesis . . . 3
- 1.3 Approach . . . 4
- 1.4 Organization of this Thesis . . . 4

2 Target Platform 7
- 2.1 Wireless Sensor Networks . . . 7
- 2.2 MICAz . . . 8
- 2.3 AVR Micro Controllers . . . 8
  - 2.3.1 Characteristics of the ATmega128L . . . 9
  - 2.3.2 AVR Instruction Set . . . 9
  - 2.3.3 Memory Map . . . 12
- 2.4 Energy Consumption of the ATmega128L . . . 13
- 2.5 Used Tools . . . 16

3 Cryptography 19
- 3.1 Symetric vs. Asymetric Cryptography . . . 20
- 3.2 Mathematical Background of ECC . . . 21
- 3.3 Related Work . . . 23

4 Elliptic Curve Cryptography on Constrained Devices 25
- 4.1 Fundamentals of ECC . . . 25
- 4.2 Elliptic Curve Optimization . . . 26
  - 4.2.1 Point Doubling . . . 27

4.2.2 Point Addition 29
4.2.3 Scalar Point Multiplication 31
4.3 Primefield Arithmetic Optimization 31
4.3.1 Operand scanning multiplication 32
4.3.2 Product scanning multiplication 32
4.3.3 Hybrid Multiplication 33
4.3.4 Inversion 36
4.3.5 Reduction 37
4.3.6 Bisection 37

5 Implementation of SECP160r1 on ATmega128L 39
5.1 Multiplication 39
5.1.1 Schoolbook Multiplication 40
5.1.2 Theoretical Minimum Effort for Modular Multiplication . . . 41
5.1.3 Bottlenecks 42
5.1.4 Operand Scanning Strategy and Carry Handling 43
5.1.5 Hybrid Multiplication and SRAM Access 46
5.2 Modular Multiplication with Column Width 10 49
5.2.1 Adapting the Row Wise Multiplication to the Target Platform 49
5.2.2 Secure Carry Add 51
5.2.3 Use Add With Carry Instruction 54
5.2.4 Discussion of the implementation with d=10 56
5.3 Modular Multiplication with Column Width 5 56
5.3.1 Pipelining EIBs to a Full Row 57
5.3.2 Pipelining Rows to a Full Column 59
5.3.3 Discussion of the Implementation with d=5 60
5.3.4 Loops 62
5.4 Reduction 62
5.5 Bisection 65
5.6 Addition and Subtraction 65
5.7 Inversion 66

6 Results 67
6.1 Speed 67
6.2 Energy 69

7 Summary and Future Work 71
7.1 Summary 71
7.2 Future Work 72

Bibliography 75

A Source Code 79

List of Figures

1.1 This Thesis' Focus 1

2.1 MICAz, [Incb] 8
2.2 ATmega128L Memory Map 14

3.1 From Cryptology to ECC 19
3.2 Elliptic Curve with $a = -7$ and $b = 11$ 22

4.1 The three Layers of an ECC-system 26

5.1 Elementary Instruction Block (EIB) 40
5.2 Row Wise Multiplication 44
5.3 Elementary Instruction Blocks Overlap Each Other 45
5.4 Column Wise Multiplication 47
5.5 160-bit Hybrid Multiplication on ATmega128L with $d = 5$ 47
5.6 UV Carry Handling 51
5.7 Save Carry Add 52
5.8 Carry Addition Scheme of an Unrolled Column 52
5.9 Handling Carry Bits by Reordering the Additions 55
5.10 Pipelining Elementary Instruction Blocks 57
5.11 Unrolled Column: Pipelining five Rows for Efficient Carry Handling 59
5.12 Reduction of a 160-bit Integer 64

List of Tables

2.1 Instruction Set Summary of the ATmega128L 10
2.2 Energy Consumption of the ATmega128L at **4 MHz** 15
2.3 Energy Consumption of the ATmega128L at **7.37 MHz** 16

5.1 SRAM and Register Requirements of the Hybrid Multiplication . . . 48
5.2 Required Instructions and Clock Cycles for a 160-bit Multiplication with $d = 10$ 54
5.3 Required Instructions and Clock Cycles for a 160-bit Multiplication with $d = 5$ 61
5.4 Reduction of a 320-bit Integer 63

6.1 Comparison of Executed Instruction for one 160-bit Multiplication . 68
6.2 Comparison of Energy Consumption 70

List of Algorithms

1 Point doubling ($y^2 = x^3 - 3x + b$ in Jacobian projective coordinates) 28
2 Point addition ($y^2 = x^3 - 3x + b$ in affine-Jacobian coordinates) . . . 30
3 Left-to-right binary method for point multiplication 31
4 Operand Scanning Multiplication 32
5 Product Scanning Multiplication 33
6 Hybrid multiplication 35
7 Binary algorithm for inversion in $\mathbb{F}_p$ 36
8 Bisection with shift in $\mathbb{F}_p$ 38

1 Introduction

The introduction will first motivate the work. Subsequently the aim of the thesis and the approach are presented. Finally the organization of this work is presented.

1.1 Motivation

The terms *ubiquitous* and *pervasive computing* designate the penetration of our everyday life with intelligent devices. Pervasive applications can be divided into active and passive applications, see Figure 1.1.

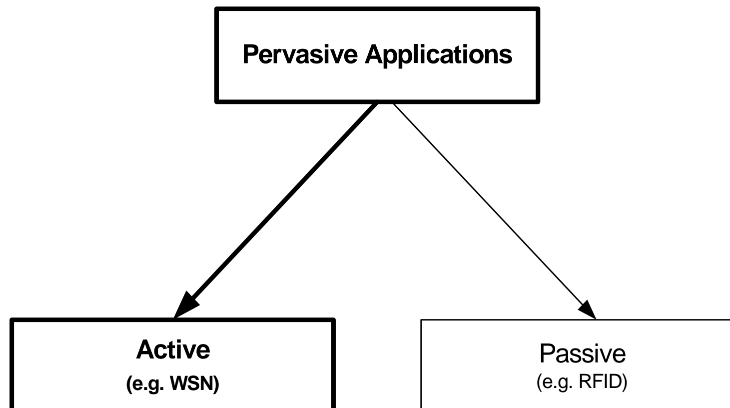

Figure 1.1: This Thesis' Focus

The main difference is that active applications do have an energy supply, whereby passive applications solely rely on the energy provided by the electromagnetic field. Two famous examples for pervasive applications are the RFID-tags (passive) and *Wireless Sensor Networks* (WSN, active). In this work we will focus on active applications, i.e. WSN.

WSN will play a fundamental role in enabling the vision of penetration of our everyday life with intelligent devices. WSNs consist of many tiny and smart devices, referred to as nodes, which typically combine an 8-bit processor with memory, sensors, radio unit, and power supply. Nodes can collect data, as they can be equipped with different kinds of sensor units, to measure temperature, humidity or electromagnetic radiation, to name just a few. The integrated radio unit allows the nodes to submit the collected data. Therefore and because they are cheap and very flexible in use the nodes are most interesting in military, agricultural, or the medical domains (e.g., for the health monitoring of human beings in medical scenarios). Obviously the data collected and transmitted by nodes can be critical concerning life, economy and infrastructure.

The nodes' wide field of application stems from their flexibility, their small size and the fact that they are cheap. This is only possible because the devices are very constrained in terms of memory, computing power, and energy supply. Since battery powered devices have a limited amount of energy, the crucial characteristic in the area of WSNs is energy consumption. The lifetime of a WSN is directly proportional to its energy efficiency, i.e., the less energy is consumed by applications the longer the batteries will last.

WSNs face major security problems because the communication is wirelessly and the devices are often easy to access. An adversary can easily eavesdrop on communication or simply steal a node. Since sensor nodes are usually not tamper-resistant, an adversary can often read out any content that is stored on the node. This constitutes a dangerous thread with the complete penetration of our lives with intelligent devices: data collected by the nodes can be read by an attacker. Data-security commissioners sound the alarm bells because of the realization that data of high economic importance can be intercepted at will. In addition, bringing harmful Motes into the network is dangerous as well: false data can be injected or measured data can be falsified during the transmission. In the automobile industry extremely sensitive applications exist as well, for example when adjusting parameters for the engine control. As motes handle secret data with economic value, a new form of industrial espionage could develop in this way.

Therefore, security is a key requirement for the success of *ubiquitous* and/or *pervasive computing* and the associated benefits of comfort and security. The communication between nodes and the measured data can be secured by using a cryptographic system. Usually a cryptographic system will need memory to store keys and use a lot of CPU power. Due to the constrained hardware, symmetric algorithms are generally preferable to asymmetric algorithms in the field of WSNs, because they are more efficient in terms of energy consumption, memory requirements, and CPU usage.

However, when symmetric algorithms are used, two problems arise: (1) key distribution and (2) number of stored keys. When individual keys are used in a WSN with n nodes, each node has to store $(n-1)$ keys. This has good resilience[1] properties but obviously scales badly and is especially unsuitable for large WSNs. Moreover, perfect forward secrecy is not given after a node's key has been compromised. When one single symmetric key is used, memory requirement is greatly reduced, but at the same time this is not resilient anymore. To cope with this problem many probabilistic key distribution schemes for symmetric algorithms have been proposed [EG02, CPS03, DDHV]. In general these approaches either need predistributed keys, which means a higher configuration effort before deployment, or they produce much traffic overhead. As the nodes radio consumes much more energy than its CPU the traffic should be minimized wherever possible.

A public key based system has an advantage concerning these problems. Its asymmetric algorithms are very valuable for key establishment and authentication in WSN. Public key cryptography can be used stand alone or just for special purposes like key establishment or signature generation in combination with a symmetric algorithm for encryption and decryption. As asymmetric algorithms are less efficient concerning energy consumption on constrained devices than symmetric algorithms, an accurate choice of optimizations is most important. To keep the system secure it should also stick to standard parameters.

1.2 Aim of this Thesis

In this thesis a highly speed optimized implementation of an asymmetric cryptographic system for 8-bit micro controllers will be presented and compared to existing systems. The thesis has two main goals. First of all the implementation shall compete with existing implementations in terms of speed and thus energy consumption. This is most important, because securing the WSN should not dramatically lower the nodes' lifetime. Furthermore the technique to achieve this shall be explained in detail.

Second the implementation is planned to become an open source project after the thesis is finished. The community shall be able to use and continue the work on securing WSNs by also using efficient public key cryptography.

[1] i.e. in case a few nodes are compromised, a security scheme should still protect against the attackers [WAR06].

1.3 Approach

To achieve a fast running cryptographic asymmetric system, first the existing asymmetric algorithms will be analyzed. Whereby only those considered as secure will be compared with others. An appropriate mathematical model will then be chosen with respect to the hardware constraints in terms of energy, memory, and CPU power.

The system will be divided into different layers for this problem: protocol, curve arithmetic, and prime field arithmetic. Though the layers can be implemented separately, the protocol and curve arithmetic layer optimization techniques are expected to vary widely for different applications. Therefore, to achieve both aims we focus on an efficient prime field arithmetic, because low-level optimizations will speed up any application based on *Elliptic Curves* (EC). As we will point out in Chapter 4 this is not true for optimizations on other layers. Furthermore optimizations on the prime field arithmetic layer enable easy potability to other implementations. This way future implementations for other protocols can reuse this implementation. To apply the algorithms the best possible way for the target hardware the bottom layer will be completely implemented in assembly. The main optimization goal is speed rather than code size. To proof functionality a running system will be implemented in C. It will include some basic optimization techniques that may be reused and expanded for specific protocols.

1.4 Organization of this Thesis

The remainder of this work is organized as follows: in Chapter 1 the aim of this thesis is motivated and an overview is given. Subsequently, in Chapter 2 we give an introduction to the target platform, i.e. the MICAz and its micro controller ATmega128L. Furthermore the energy consumption is considered. The chapter ends with the description of the tool chain. Subsequently, in Chapter 3 an introduction to cryptography is given, that includes security services and mathematical background. The chapter closes with an overview of the related work. Subsequently, in Chapter 4 we introduce some fundamentals concerning optimization and present all algorithms we use in this work, i.e. algorithms used to perform the modular arithmetic as well as algorithms used to perform curve arithmetic. In Chapter 5 we then present details about the implementation. The chapter starts with a general analysis of the schoolbook algorithm, where a definition for the *minimum effort* is done, as well as a bottleneck analysis. In the following different variants of *schoolbook multiplication* are discussed with respect to the bottlenecks *carry handling* and *SRAM access*. The

chapter continues with implementation details of the *hybrid multiplication* using a column width of 10. The analysis of this implementation is followed by the implementation of the *hybrid multiplication* with a column width of 5. This very efficient implementation is then discussed and options for applying loops to the implementation are described further on. Furthermore we provide implementation details of the modular operations reduction, bisection, addition, subtraction, and inversion. In chapter 6 we summarize our results. Finally, we conclude in Chapter 7 and give an outlook to future work.

2 Target Platform

The chapter starts with an introduction to *Wireless Sensor Networks* (WSN). Subsequently the node MICAz is introduced. The MICAz's micros controller ATmega128L is discussed in the following section supplying an overview of characteristics, instructions set, and memory map. Subsequently energy consumption is analyzed. The chapter closes with an overview of the used tool chain for the implementation.

2.1 Wireless Sensor Networks

WSN are built from tiny, smart, and cheap devices called nodes. These attributes open applications amongst others in agriculture, traffic system, civil life, military, and nature preserves. Spotting free parking areas, individual plant monitoring, monitoring bridges and houses in earthquake vulnerable areas are just a few examples of what can be done with WSNs, [HC02]

Their size and low-cost feature constrains them in memory, CPU power, and energy supply. For the envisioned applications of WSNs, up to tens of thousands of smart, but battery powered devices are required. WSN typically form a wireless ad-hoc network with multi-hop communication patterns, [MM04]. Ideally the batteries should last for several months up to years. Therefore, low power consumption is a crucial requirement for any application running on these nodes. Sending and receiving of messages is by far the most energy consuming task on the nodes [HSW+00], therefore the traffic should be minimized wherever possible. Furthermore, the energy consumption of an application is mainly determined by its execution time. A rule-of-thumb is: the shorter the processing time of an algorithm, the lower its energy consumption.

2.2 MICAz

Nowadays, the de-facto standard sensor nodes for researchers are the so-called MICA motes [Incb] developed at the University of California, Berkeley [HC02]. Figure 2.1 shows a MICAz [Inca] with standard antenna. MICA motes are produced and shipped by Crossbow Technology [Incc]. The MPR2400, also called MICAz is

Figure 2.1: MICAz, [Incb]

the latest generation of motes from Crossbow Technology. It comprises of:

- ATmega128L micro controller
- Chipcon CC2420 radio frequency transceiver
- AT45DB041 4-Mbit serial flash

The ATmega128L [Cora] is an 8-bit AVR micro controller by Atmel Corporation, see Section 2.3. The radio interface is a ZigBee ready, IEEE 802.15.4 compliant, frequency transceiver in the range of 2400 MHz to 2483.5 MHz. All motes feature Atmel AT45DB041 4-Mbit serial flash, which is connected to one of the USART on the ATMega128L. It can be used for storing data, measurements, and other user-defined information. MICA2 application software and sensor boards are compatible with the MPR2400.

2.3 AVR Micro Controllers

The AVR micro controller product line is designed for low power consumption and high performance: according to [Corb] it fits a Harvard architecture with true single

cycle execution, offers 32 general purpose registers, and performs 20 MIPS at 20Mhz. The AVR micro controller product line can operate from 1.8 volts to 5.5 volts, provides a sleep controller with a variety of operation modes, fast wake-up from low power modes, and the operation frequency is software controlled. However, contrarily to description given by Atmel Corporation the ATmega128L does execute some non-single-cycle-instruction as most of the AVR do, e.g. an 8-bit requires two cycles. This section will describe the hardware characteristics of the ATmega128L [Cora] followed by a short overview of the instruction set and finally presents its memory map.

2.3.1 Characteristics of the ATmega128L

The ATmega128L comes with 128 KB of in-system reprogrammable flash, 4 KB EEPROM, 4 KB internal SRAM, and can handle up to 64 KB optional external memory space. The processor knows 133 instructions, including a 2-cycle 8-bit multiply, and achieves throughputs approaching one MIPS per MHz allowing the system designer to optimize power consumption versus processing speed. The ATmega128L, which is integrated in the MICAz, runs between 0 MHz and 8 MHz. Furthermore the micro controller provides two 8-bit and two 16-bit counters and one real time counter with separate oscillator. Six sleep modes (idle, ADC noise reduction, power-save, power-down, standby) lower the energy consumption, which is valuable, since the MICA motes are supposed to be idle about 99% of their life time.

2.3.2 AVR Instruction Set

Out of the instruction set we summarize the most important ones for this thesis in Table 2.1. The nomenclature used in Table 2.1 equals the one used by Atmel Corporation. A complete overview and a detailed description of all assembly instructions including the nomenclature can be found in the AVRStudios [Corc] *Assembly help* and in the AVR Assembly User Guide [Cord].

Instruction set nomenclature concerning the *Status Register* (SREG) flags:

- C : Carry flag in status register
- Z : Zero flag in status register
- N : Negative flag in status register

Instruction	Description	Used Flags	Cycles
ADD Rd, Rr	Add two Registers	Z,C,N,V,H	1
ADC Rd, Rr	Add two Registers with carry	Z,C,N,V,H	1
SUB Rd, Rr	Subtracts two Registers	Z,C,N,V,H	1
SBC Rd, Rr	Subtracts two Registers with carry	Z,C,N,V,H	1
MOV Rd, Rr	Move 8-bit word	none	1
MOVW Rd+1:Rd,Rr+1:Rr	Move 16-bit word	none	1
MUL Rd, Rr	Multiply Unsigned	Z,C	2
LDI Rd, K	Load immediate	none	2
LDS Rd, k	Load direct	none	2
LD Rd, X+	Load indirect with post increment	none	2
STS k, Rd	Store direct to SRAM	none	2
ST X+, Rd	Store indirect with post increment	none	2

Table 2.1: Instruction Set Summary of the ATmega128L

- V : Two's complement overflow indicator
- H : Half carry flag in the status register
- S : Sign bit, S = N$\oplus$V
- T : Bit copy storage
- I : Global interrupt enable

The *carry flag* (C) is essential for the implementation contrary to the other flags. The carry is set to 1, if an operation's destination register is too small for the result, otherwise it is set to 0. The exact rule how a flag is set or reset is defined by the instruction, which can be found in the AVRStudios [Corc] *Assembly help*.

Instruction set nomenclature concerning *Registers and Operands*:

- Rd : Destination (and source) register
- Rr : Source register
- K : Constant data
- k : Constant address
- X : Indirect address register (X=R27:R26, Y=R29:R28 and Z=R31:R30)

The registers symbolized by **Rd** can be in the range of R0-R31 or R16-R31, depending on the instruction while the registers symbolized by **Rr** are always in the range of R0-R31. That means not every instruction can be performed on any register, e.g. ADD falls in the first category and can be performed with any register while LD falls in the second category and can only be performed on R16-R31. **K** is a data constant and can hold any number that fits into a 1-byte binary number (0-255) while the address constant **k**'s value range depends on the instruction. The ATmega128L can operate with three 16-bit pointer (**X**, **Y** and **Z**) to address the SRAM comprised of two 8-bit registers, i.e registers R26 and R27 for **X**, R28 and R29 for **Y**, and R30 and R31 for **Z**.

In Table 2.1 each row consists of the instruction syntax (mnemonic and operands), a short description, the flags that are set in the SREG, and finally the number of clock cycles required. The table starts with arithmetic addition and subtraction,

both with usage of carry (ADC, SBC) and without (ADD, SUB). These commands require one cycle to be performed and they are all usable with any register. Next are the transfer instructions MOV and MOVW, which move an 8-bit word or a 16-bit word and consume one cycle. The MOVW instruction can on the one hand move two registers in one clock cycle and on the other hand enforces the source and destination registers (as shown in the syntax mnemonic) to be even. That makes the instruction extremely useful, because sometimes registers values need to be moved, as not every instruction can be performed on any register. The following instruction (MUL) performs a signed or unsigned 8-bit multiplication. The 16-bit result is always saved to R0 and R1. The repeated execution of this instruction is circumstantial due to the fixed destination registers. Finally three load and two store instruction are listed. Load and store instruction exist as direct (LDS, STS), and indirect (LD, ST) variants. A direct store or load uses a constant as address while the indirect store and load instructions use one of the address pointers (**X**, **Y** or **Z**). Both variants require two clock cycles. The indirect loads and stores can be executed with additional post-increment (LD Rd, +X), and pre-decrement (LD Rd, -X) without increasing the execution time. The LDI instruction loads immediately in two clock cycles a constant to a register.

The most expensive instructions that will be used are multiplications and memory loads and stores. The choice of algorithms should respect that the potential for saving clock cycles and thus energy is remarkably higher for these instructions. Besides this, classical multiplication-memory-trade-offs, generally working with precalculated values, might not lead to time saving, because an 8-bit multiplication takes the same time as an SRAM access.

2.3.3 Memory Map

Figure 2.2 shows the memory map of the ATmega128L in hexadecimal address format. The address space from 0x0000 to 0x0100 contains the 32 *general purpose registers* followed by the *special function registers*, [Cora]. The adjacent address space is the 4 KB SRAM that has its maximum address at 0x1100. Optionally, the external SRAM follows - going up to address 0xFFFF.

The internal SRAM is separated into four parts: First is *data*, which contains all static variables that are initialized unequal to zero. Next is the HEAP, its size is dynamic and depends on the amount of dynamically allocated memory at any time. The HEAP grows with increasing address. The STACK starts at the top of the internal SRAM. This data segment is, similar to the HEAP, dynamic, but contrary to the HEAP it grows with decreasing addresses towards the HEAP.

Allocating variables expands the HEAP during program execution while push operations expand the STACK. Both actions are consuming memory. If the micro controller runs out of memory the STACK overwrites the HEAP or the other way round. As memory is constrained on micro controllers variables should be used with care.

Static variables always remain in the SRAM. Instead of declaring constants as static variables, they can be sourced out to the flash memory. Two possibilities are available, either integrate the variable in the program code, by using the value with the LDI instruction or by declaring the variable as flash memory content using the .DB assembly directive. The former solution is feasible for constants that are always used with the same piece of code and not used elsewhere. This solution may also be used in time critical parts, as it is even faster than keeping that variable in SRAM. On the other hand a lot of flash memory may be consumed compared to other solutions.

A short example for the first option:

```
1 mul_a:
2 .db $00,$00,$00,$00,$E1,$9F,$CF,$13,$6B,$97,$37,$26,$8A,$08,$F8,$35
3   ldi ZH,HIGH(mul_a*2)
4   ldi ZL,LOW(mul_a*2)
5   //eventually apply an offset here
6   lpm r26,Z+
7   lpm r27,Z+
8   lpm r28,Z+
9   lpm r29,Z+
```

Declaring variables as part of flash memory is a feasible solution if the variable is a non-time-critical part of the program or used frequently. The variable can then be loaded to the registers using the ELPM instruction and one of the pointers. It is also possible to store a copy to SRAM for later use, as SRAM is 50% faster than flash memory.

For an example for the second option we referr to Section 5.5. This solution is feasible if the value is used only once, or in time critical situations.

2.4 Energy Consumption of the ATmega128L

There are many sources available which state the energy consumption of the ATmega128L. The data sheet of the ATmega128L [Cora] states a current drain of 19 mA at 7.37 MHz with a power supply of 5 V or 5.5 mA at 4 MHz and 3 V. The MICAz data sheet [Inca] states a power supply between 2.7 V and 3.3 V with a current

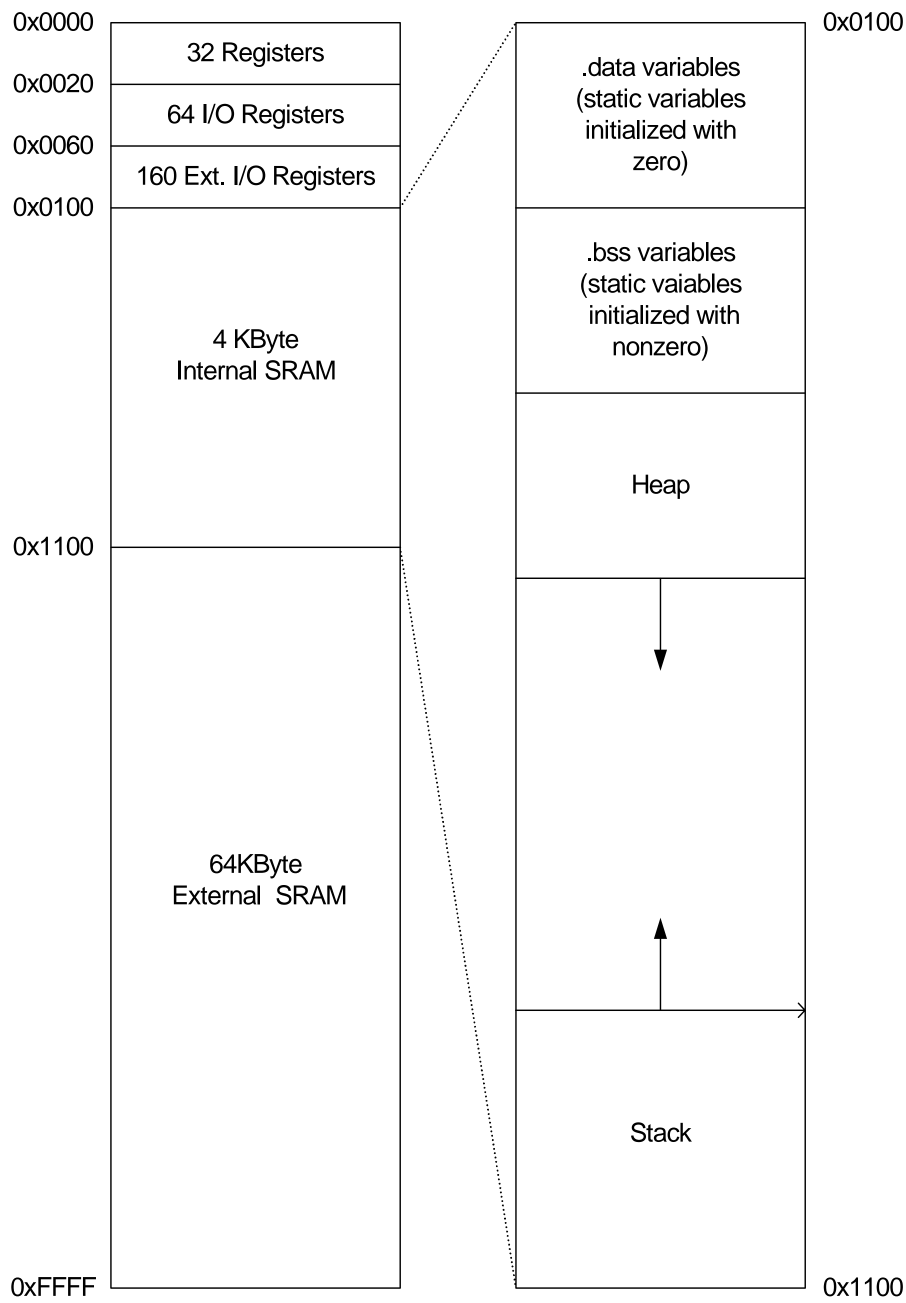

Figure 2.2: ATmega128L Memory Map

drain of 8 mA. On the other hand the user manual "Mote Processor Radio (MPR) platforms and Mote Interface Boards (MIB)" [Incb] gives more detailed information about energy consumption of the ATmega128L on all available types of Motes. According to this manual an ATmega128L on a MICAz with a power supply between 2.7 V and 3.6 V consumes 12 mA at 7.37 MHz and 6 mA at 4 MHz. Surprisingly on the following page of the same source, there is an example in which the current drain is stated as 8 mA.

As manufacturer information vary, the measurements found in [WGE+05] are considered to be helpful. Unfortunately, the analyses were done for a MICA2 clocked at 4 MHz, while this project is supposed to run at 7.37 MHz. In [WGE+05] the current drawn from the Motes power supply was measured and from these results the energy consumption of the micro controller was approximated. The authors verified their result with another more accurate approach with an oscilloscope and a sense resistor, which showed up an accuracy of 5%. Tables 2.2 and 2.3 list all information. In addition the resulting energy consumption was added for each case as well as the consumed energy per clock cycle.

Source	**Power Supply**	**Current**	**Energy**	**Consumed Energy per Clock Cycle**
[Cora]	3 V	5.5 mA	16,5 mW	4.13 nWs
[Inca]	3 V	8 mA	24 mW	6.00 nWs
[Incb]	3 V	6 mA	18 mW	4.50 nWs
[WGE+05]	3 V	4.5 mA	13.8 mW	**3.45 nWs**

Table 2.2: Energy Consumption of the ATmega128L at **4 MHz**

The measured energy consumption of 3.45 nWs at 4 Mhz seems to be most reliable, because the authors checked their results with a more accurate system later on and both measurements had a 5% difference. At a clock rate of 7.37 MHz, no such measurement was provided, and the information from the data sheets differ enormously. This might be caused by the micro controller running at 3 V instead of 5 V, where the latter is the specification by Atmel Corporation.

The measured energy consumption of 4.88 nWs at 7.37 Mhz seems to be most reliable for the same reasons as above. We will use this value to estimate energy consumptions in the remainder of this thesis.

Source	Power Supply	Current	Energy	Consumed Power per Clock Cycle
[Cora]	5 V	19 mA	95 mW	12.89 nWs
[Inca]	-	-	-	-
[Incb]	3 V	12 mA	36 mW	**4.88 nWs**
[WGE+05]	-	-	-	-

Table 2.3: Energy Consumption of the ATmega128L at **7.37 MHz**

2.5 Used Tools

A large variety of tools exist for implementing for AVR-micro controller [Corb]. A set of tools must be selected and configured. This will be at least an editor, a compiler, a linker, and a simulator. Tool chains usually are more flexible but need more configuration as compared to stand alone solutions.

As it is most important for this project that the implemented algorithms perfectly fit to the processors architecture, the simulator must simulate the micro controller, especially the CPU. The currently available simulators for TinyOS [HSW+00] respectively NES-C [GLvB+03] do not satisfactorily meet this requirement. The AVRStudio [Corc] is a complete *Software Development Kit* (SDK) developed by Atmel. It includes a suitable simulator for Win32 that is used for the implementation. The AVRStudio can debug assembly code and understands debug information in dwarf-2, coff and extcoff format. The WinAVR [Win] tool chain upgrades the possibility to create and compile projects written in C. Writing and compiling mixed language projects is also possible with this tool chain. Unfortunately it could not be managed to hand over parameters from the AVRStudio to the compiler provided by WinAVR, such that studio-simulator could handle the generated debug informations. E.g. jumping into an assembly function that was called from C code resulted in disassembled code view. For this reason the WinAVR tool chain was directly used and AVRStudio remained as a plain simulator. The code had to be patched, to fit the GNU-assembly [GB] (GAS). The debug problem could finally be solved by changing the debug format from DWARF-2 to COFF. Further debug problems with structs and arrays lead to the EXTCOFF format.

The WinAVR project provides a powerful tool chain very suitable for the project presented in this thesis. The used components are:

- avr-libc
- GNU Binutils
- GNU GCC
- Mfile

The *avr-libc* [proa] is a C library for AVR that was designed to conform the standard C library as described by the ANSI X3.159-1989 and ISO/IEC 9899:1990 ("ANSI-C") standard, as well as parts of their successor ISO/IEC 9899:1999 ("C99"). It also includes extensions that are conform to other standards or are AVR specific e.g. it provides startup code that most applications will need. Especially for time critical applications a basic assembly support is also included.

The *GNU Binutils* [GB] and the *GNU Compiler Collection* [Prob] (GCC) contain among others the AVR-ASSEMBLY and the AVR-GCC for compiling assembly and C. The assembly and linker are usually not invoked manually, but rather using the C compiler front end (avr-gcc) that in turn will call the assembly and linker as required. This procedure is done with the help of a makefile containing all parameters.

The MFILE tool serves with a graphical front end and provides different makefile templates for common scenarios of calling compiler and linker by the C compiler frontend. Furthermore it offers different formats of debug information to choose from. In addition to these components the WINAVR software packet also provides a standalone version of the PROGRAMMERS NOTEPAD2 [oss] with predefined makefile calls for commonly used parameters working e.g with makefiles created by MFILE.

With this tool chain the source code can be organized in a project using PROGRAMMERS NOTEPAD2 while compiler and linker are callable with a single hot key. Thus an integration similar to that when working with AVRSTUDIO is achieved. For simulation a window switch to the AVRSTUDIO is necessary. All tools but the AVRSTUDIO currently are available for different kinds of platforms. Other simulators like AVR-GDB and SIMULAVR can easily be integrated in the tool chain when operating on platforms other than win32, but most likely an according debug format has to be chosen in that case, e.g. the more modern DWARF-2 format. The AVR tool chain includes many more tools suitable for programming micro controllers, on chip debugging and more.

3 Cryptography

Cryptography is the study of keeping messages secure. It can be divided in to symmetric and asymmetric techniques. Besides cryptography there exists cryptanalysis, which is the study of breaking secured messages. The topic for both is cryptology. This Chapter starts with an introduction to cryptography and will also cover the required mathematical background. Subsequently some cryptographic schemes are explained that are suitable for constrained devices. Finally an overview of related work is given. In Figure 3.1 the location of Elliptic Curve Cryptography (ECC) in the wide field of cryptology is shown. For a general introduction to cryptology the reader is referred to [Sch96].

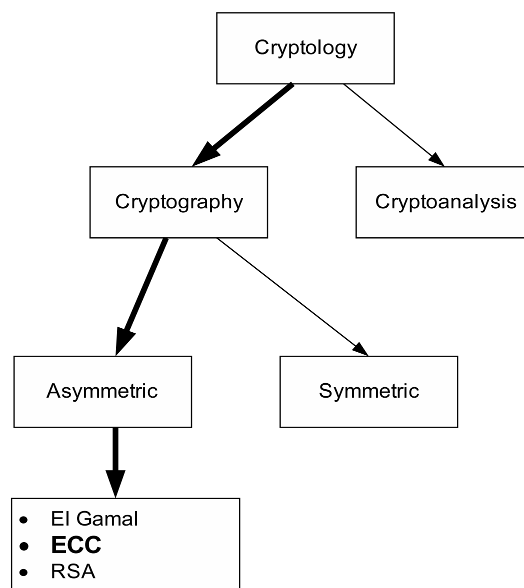

Figure 3.1: From Cryptology to ECC

3.1 Symetric vs. Asymetric Cryptography

The four main security issues are:

- Confidentiality
- Authentication
- Integrity
- Non-repudiation

Confidentiality means that persons without permission cannot read a message. *Authentication* means that the origin of a message can be reliable ascertained. An *integer* message cannot be modified unnoticed. *Non-repudiation* has the intent to prevent the sender of a message from falsely denying to have sent the message. For further details the reader is referred to [Sch96]. All these security services can be achieved by cryptography.

When securing a channel or encrypting messages there are generally two approaches: symmetric ciphers and asymmetric ciphers. Both, symmetric and asymmetric ciphers can be used for encryption. While in symmetric systems the key must be secret and authentic, in asymmetric systems the exchanged key material has only to be authentic, [HMV04]. Symmetric systems are called *private key systems*, whereas asymmetric systems are called *public key systems*. They work significantly different and have pros and cons concerning efficiency and prerequisites.

In *symmetric systems*, for decryption and encryption the same key is used. Furthermore, the entities sharing a key must be sure about the identity of the other key holder. For this reason the key must be shared through a secret and authenticated channel. This is called the *Key Distribution Problem*. Additional problems occur if a multitude of key holders is involved - as it is the case in the area of WSNs.

Another aspect is the so called *Key Management Problem*. In a network with n entities, each entity has to store $n-1$ keys to be able to communicate securely with all other entities. In large WSNs built from constrained devices this obviously scales badly. According to [HMV04] all known solutions working with symmetric keys are generally impractical. Symmetric systems are similar to a treasure box. A specific key is needed to put something into it. No one without this key can open or look into the box. The handover of the key from sender to receiver is problematic in the same way. Nobody must be allowed to copy the key and authentication is needed.

Asmmetric systems only need an authenticated channel to exchange keys. An entity's key material consists of a private key, which has to be kept secret and the corresponding public key, which can be transmitted unencrypted to any other entity. The key pair has the property that with a given public key the matching private key is "hard" to derive. Hard means that solving the corresponding computational problem is believed to be intractable. Number theoretic problems used to build public key cryptography are:

- disctrete logarithm problem which ElGamal is based on
- integer factorization problem which RSA is based on
- elliptic curve discrete logarithm problem which all ECC are based on

The advantage of an asymmetric algorithm over a symmetric one is the small number of keys to be stored. At least each device needs to store only its own private key. This makes it possible to decode data and create digital signatures or authenticate signatures. Also the key exchange to encrypt data is more easy. Only the public (not secret) key of the receiver is needed.

Compared with symmetric algorithms the asymmetric algorithms work very slow. In particular on low-power processors they are felt as not practical and are used only rarely or not at all. For this purpose special algorithms were developed, but they have to be cryptanalyzed and shown to be secure, which takes a long time, before they are suitable for protecting sensitive data or application. Elliptic curves represent a special case. The advantage of the ECC is that on one hand it is meanwhile quite well investigated and thus considered to be secure while on the other hand just a very short bit length is needed as compared to other asymmetric systems. Recall that 1024-bit RSA key length meets the security of 160 bit ECC key length. This is a ratio of 6.4 and will significantly reduce the consumed energy for key establishment.

3.2 Mathematical Background of ECC

Let E be an elliptic curve defined over a field K as shown in Figure 3.2, then a set of points can be created by a *chord-and-tangent rule* (extended addition). If P and Q are two different points, which are part of the set that intersect the elliptic curve in a straight line, there will be a third intersection on the straight line with the curve. The reflection on the x axis of the latter is called R and represents the sum of P and

Q. Doubling works in the same way, but the straight line is given by the tangent of the curve in the according point. The set of points defined by the extended addition and extended by the point ∞ forms an Abelian group.

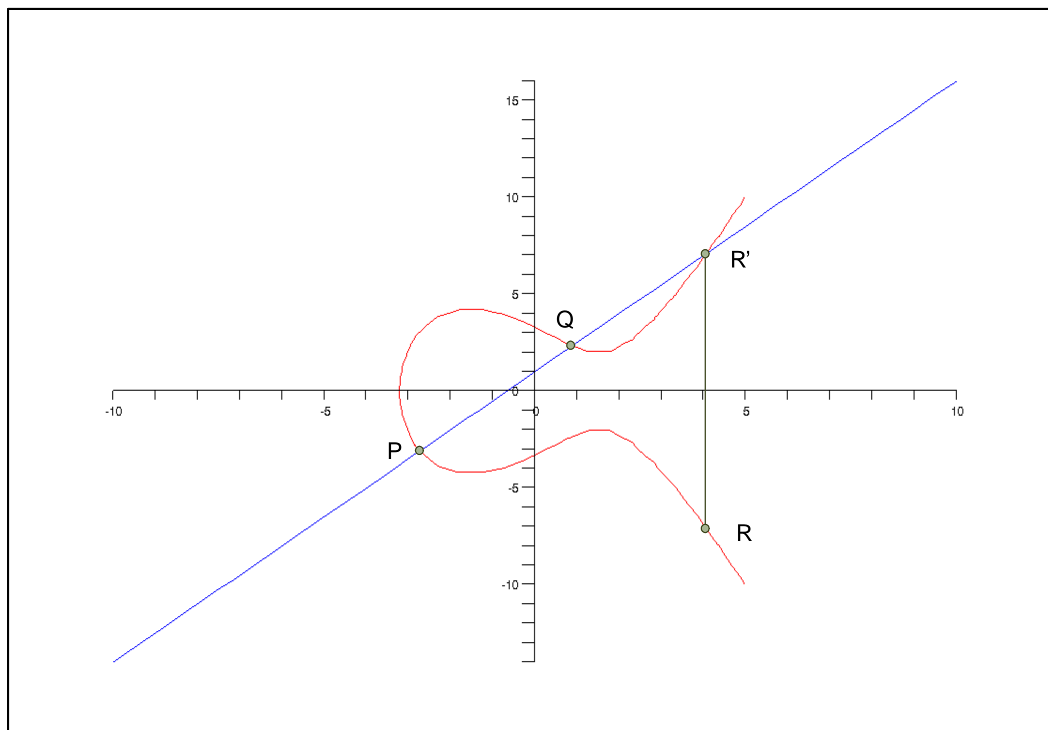

Figure 3.2: Elliptic Curve with $a = -7$ and $b = 11$

$P + P$ is referred to as $2P$. Accordingly is the scalar multiplication $P + .. + P = kP$ also referred to as point multiplication. For every Point P exists a point Q with $P = kQ$, if P is not the identity (neutral element) and the order of the elliptic curve is prime. Finding the appropriate k for a given set (Q, P) is considered to be hard and called the *elliptic curve discrete logarithm problem* (ECDLP). All ECC protocols rely on the ECDLP.

As prime fields have the potential to be implemented in software with good performance, we rely in the following on elliptic curves of the form

$$E/K : y^2 = x^3 + ax + b, char(K) \neq 2, 3 \tag{3.1}$$

The above described method for adding and doubling points on the curve has the disadvantage, that a modular inversion is required, which performs badly. By chang-

ing the coordinates system this computational intensive operation can be nearly avoided. A single inversion is required for transforming back in *affine coordinates.* In *Jacobian projective coordinates* the coordinates are transformed from $(x : y)$(affine representation) to $(X : Y : Z)$(Jacobian projective representation) by $y = Y/Z^2$, and $x = X/Z^3$. The same equations are used to transform back to *affine coordinates*, at this point the inversion is needed but only once. Applying this to Equation 3.1 results in

$$Y^2 = x^3 - 3XZ^4 + bZ^6 \tag{3.2}$$

Concerning algorithms for addition and doubling it is possible to mix up different coordinates and then pick the combination that performs best. We present the used algorithms in Section 4.2.

3.3 Related Work

Asymmetric cryptography has been long seen as too demanding for constrained devices such as sensor nodes with an 8-bit micro controller. However, there exist several implementations of asymmetric cryptographic algorithms for such micro controller. Known asymmetric systems will be briefly described and further on some of them will be used for comparison.

In [WKC+04] Watro et al. describe public-key based protocols for WSNs. In particular, they present authentication and key-agreement protocols based on RSA. The so-called TinyPK was implemented in NesC for MICAz 8-bit micro controllers. However, one RSA exponentiation with a 1024-bit key needs 14.5 seconds. Furthermore, in order to reach a security level, which is equivalent to an RSA key with a length of 1024-Bit, only 160 bits are required with ECC [Res00, LV99]. Having the limited amount of memory, computing power and energy of a typical 8-bit sensor node in mind, it seems that ECC is a much better choice for public-key cryptography for WSN rather than RSA. Since TinyPK [WKC+04] is based on the more demanding RSA algorithm and was implemented in NesC, it is not surprising that this is more than one order of magnitude slower than the fastest known implementation of a point multiplication for ECC in assembly.

EccM [MWS04] is an open source key distribution system based on ECC developed for usage in CodeBlue [oES] (Wireless Sensor Networks for Medical Care) at the University of Harvard. The system was run on an ATmega128L at 7.37 MHz but is

not based on a prime field. Furthermore no information about actual performance of curve operation are provided. The implementation can perform key generation in about 34 seconds and shared secrets can be distributed in among the same time.

In [GPW+04] N. Gura et al. present and compare fast implementations of ECC and RSA on the ATmega128L micro controller. They achieved a point multiplication on a 160-bit standard curve within 0.81 seconds. The majority (77%) of the clock cycles was required by the modular multiplication. However, the source code of this implementation is not publicly available, it is rather intellectual property of SUN Microsystems. Therefore, the results are not usable for the scientific community. Nevertheless these impressive results where furtheron analyzed for energy consumption in [WGE+05] and correlated to a node's lifetime in [PLP06]. [GPW+04] provides useful information about used techniques to speed up the prime field and the curve. The given descriptions inspired the implementation presented in this thesis.

Alternatively there is the TinyECC implementation [LN06], which may be used free of charge. TinyECC is a free software package for TinyOS that supports all SECG recommended 128-bit, 160-bit and 192-bit elliptic curve domain parameters. At the start of this thesis the runtime of the TinyECC for signature generation was 6.094s and for signature verification 12.242s. By now version 0.2 and 0.3 were released providing a significant performance gain. The signature generation is performed in 1.925s and the signature verification is performed in 2.433s. Furthermore a precomputation of 3.548s has to be done.

In summary we can say, that the fastest implementation come from SUN Microsystems and the fastest free implementation is TinyECC.

4 Elliptic Curve Cryptography on Constrained Devices

This chapter describes a selection of algorithms suitable for ECC on constraint devices. First ECC is divided into three layers, which can be implemented separately. Subsequently the used algorithms are described that implement addition, doubling and scalar multiplication on the EC. Next the algorithms used for the implementation of the prime field are described, i.e. 160-bit longterm multiplication, modular reduction, modular bisection, and modular inversion. For modular addition and modular subtraction no special algorithms are used.

4.1 Fundamentals of ECC

The basis for an efficient cryptographic system based on elliptic curves is a very efficient prime field arithmetic. As shown in Figure 4.1, a cryptographic system based on elliptic curves can be divided into three layers. The highest level actually represents the application layer. Protocols implemented here are for example ECDSA [HMV04] or EC ElGamal [HMV04]. Optimizations in this layer vary strongly, depending on the application (signature, encryption etc.) and have to be partly or completely redone for each application.

The underlying layer is the arithmetic of the elliptic curve. Most protocols are based on the multiplication of a point on the elliptic curve with an integer ($k*P$). However, optimizations at this level usually also strongly depend on the protocol layer.

Beneath the curve arithmetic lies the prime field arithmetic. Optimizations in the underlying prime field arithmetics layer will always improve the performance of the whole ECC-System, because they are layer independent. Furthermore in our special case (secp160r1 on 8-bit) more than 77% of the computing time can be applied here [GPW+04]. Therefore, a very efficient prime field arithmetic is crucial for ECC based systems on constrained devices and time critical systems.

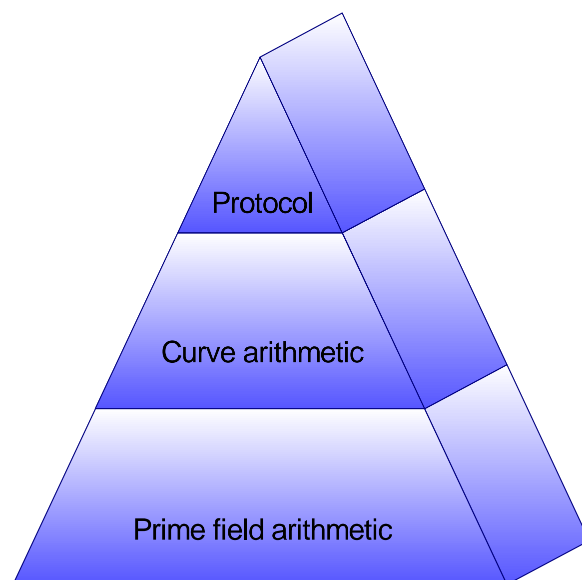

Figure 4.1: The three Layers of an ECC-system

Not focusing on a specific layer there are three general approaches to optimize: First by choosing smart parameters for the elliptic curve. Usually NIST or SECG2 standardized curves already have smart parameters. Second, by applying smart algorithms. Various possibilities do exist here. The top layer protocol used, as well as the hardware characteristics influence which algorithms perform better. An accurate choice is the challenge. Third, the algorithms need special treatment when applying them to the hardware. Best performance is achieved in this regard by using assembly.

4.2 Elliptic Curve Optimization

Several optimizations can be applied in the curve arithmetic. This section will not cover all of them, but rather focus on those, that are most important and easy to apply. The main focus lies on the prime field. The curve is optimized in terms of curve parameters and its coordinate system. Neither algorithms requiring precalculation will be adopted, nor will the curve arithmetic be implemented in assembly.

There are various algorithms for the extended addition on an elliptic curve for differ-

ent coordinates and different underlying fields. They can be optimized according to the used protocol and hardware. A good overview is given by [HMV04] and [Bro01]. Regardless of which algorithm is used, they are all based on the arithmetic of the underlying field. Especially the multiplication in the field comes at great cost in time and energy. An efficient field arithmetic is therefore the base for an efficient implementation of an ECC system.

According to [CMO98] a coordinate system mixing *Jacobian-projective coordinates* and *affine coordinates* performs pretty well. Better approaches are possible, but significantly increase the implementation effort and also the code size. On the other hand the system chosen here might be further optimized later, or in other words it is reusable. The elliptic curve provides the operation *add two points*, *double one point* and *multiply point with integer*. The add and the double algorithms can be improved if the curve parameter a of Equation 3.1 is chosen equal to -3. The curves recommended by NIST or SECG2 fit to this parameter hence this optimization can be applied. The algorithms implementing the curve operation *double* and *add* are using solely modular arithmetic of the prime field. Required operations are *addition, subtraction, multiplication, and bisection*. Bisection, though, is optional as it may be implemented by a multiplication with the modular inverse of 2, but this would perform badly, contrary to the here presented variant. The multiplication on the curve is using the EC operation *add* and *double*.

In the following three algorithms are presented that are implemented in C. Algorithm 1, [HMV04], and Algorithm 2, [HMV04] are optimized for speed while keeping the amount of needed temporary variables as low as possible. Unfortunately in the implementation of the *hybrid multiplication* the pointer to the first operand must not equal the pointer for the result. This creates two issues: First, during the implementation it has to be respected that multiplications of the form `t1<-t1·t2` should call the multiplication like `t1<-t2·t1`, which is logically the same, but solves the before mentioned problem. Second, squaring of the form `t1<-t1·t1` is not possible. To cope with this problem the notation of the Algorithms from [HMV04] where slightly modified, but without influencing the performance. The operations were not changed at all, but the variables that store the intermediate results were switched some times to avoid impossible function calls during the implementation. Algorithm 3, [HMV04] was also implemented in C, it relies on Algorithm 1 and Algorithm 2.

4.2.1 Point Doubling

The point doubling algorithm presented here is a modified version from [HMV04]. It is implemented for the curve secp160r1, see Chapter 5. The algorithm requires at

least the modular operations *addition*, and *multiplication*. For performance reasons *subtraction*, and *bisection* were also implemented. A better performance might be reached with a *modular sparing* operation, because if realized as an extra operation it can optimized and achieve better performance. In general the algorithm needs to execute four modular multiplication and four modular squaring, whereby the later are realized as multiplications in this case.

The algorithm requires as input a point on the elliptic curve in Jacobian projective coordinates. Note that if an affine point is given, the Jacobian projective representations can be derived by $(X, Y, Z) = (x, y, 1)$. If the input equals the point at infinity the latter will be returned as result. In any other case the algorithm returns the double of the input in Jacobian projective coordinates.

Algorithm 1 Point doubling ($y^2 = x^3 - 3x + b$ in Jacobian projective coordinates)

Require: $P = (X_1 : Y_1 : Z_1)$ in Jacobian coordinates on $E/K : y^2 = x^3 - 3x + b$.
Ensure: $2 \cdots P = (X_3 : Y_3 : Z_3)$ in Jacobian coordinates.
1: **if** $P = \infty$ **then**
2: **return** (∞)
3: **end if**
4: $T_1 \leftarrow Z_1^2$
5: $T_2 \leftarrow X_1 - T_1$
6: $T_1 \leftarrow X_1 + T_1$
7: $T_3 \leftarrow T_2 \cdot T_1$
8: $T_2 \leftarrow 3 \cdot T_3$
9: $Y_3 \leftarrow 2 \cdot Y_1$
10: $Z_3 \leftarrow Y_3 \cdot Z_1$
11: $X_3 \leftarrow Y_3^2$
12: $T_3 \leftarrow X_3 \cdot X_1$
13: $Y_3 \leftarrow X_3^2$
14: $Y_3 \leftarrow Y_3/2$
15: $X_3 \leftarrow T_2^2$
16: $T_1 \leftarrow 2 \cdot T_3$
17: $X_3 \leftarrow X_3 - T_1$
18: $T_1 \leftarrow T_3 - X_3$
19: $T_1 \leftarrow T_1 \cdot T_2$
20: $Y_3 \leftarrow T_1 - Y_3$
21: **return** $(X_3 : Y_3 : Z_3)$

4.2.2 Point Addition

The point addition performs an addition of two points of an elliptic curve in mixed coordinates. That means as input one point is given in affine coordinates while the other point is given in Jacobian projective coordinates. The algorithm calculates then $P + Q$ and returns the result in Jacobian projective coordinates.

Furthermore some special cases must be checked. The algorithm checks if one of the input points is the point at infinity if so the point at infinity will be returned as output. The algorithm furthermore checks if the points are equal in affine coordinates, in which case the point double algorithm is called, which in turn will calculate and return the result. P transformed to affine representation would be $(X_1/Z_1^3, X_2^2)$, which must not equal $Q = (x_2, y_2)$. The two equation for the two ordinates can be transformed to $X_1 = Z_1^2 \cdot x_2$ and $Z_1^3 \cdot y_2 = Y_1$. T_1 and T_2 in line eleven and twelve represent $T_1 = Z_1^2 \cdot x_2 - X_1$ and $T_2 = Z_1^3 \cdot y_2 - Y_1$ which means, if both equal zero, that $P = Q$.

Algorithm 2 Point addition ($y^2 = x^3 - 3x + b$ in affine-Jacobian coordinates)

Require: $P = (X_1 : Y_1 : Z_1)$ in Jacobian coordinates, $Q = (x_2 : y_2)$ in affine coordinates on $E/K : y^2 = x^3 - 3x + b$
Ensure: $P + Q = (X_3 : Y_3 : Z_3)$ in Jacobian coordinates

1: **if** $Q = \infty$ **then**
2: **return** $(X_1 : Y_1 : Z_1)$
3: **end if**
4: **if** $P = \infty$ **then**
5: **return** $(x_2 : y_2 : 1)$
6: **end if**
7: $T_1 \leftarrow Z_1^2$
8: $T_2 \leftarrow T_1 \cdot Z_1$
9: $T_3 \leftarrow T_1 \cdot x_2$
10: $T_4 \leftarrow T_2 \cdot y_2$
11: $T_1 \leftarrow T_3 - X_1$
12: $T_2 \leftarrow T_4 - Y_1$
13: **if** $T_1 = 0$ **then**
14: **if** $T_2 = 0$ **then**
15: use Algorithm 1 to compute $(X_3 : Y_3 : Z_3) = 2 \cdots (x_2 : y_2 : 1)$
16: **return** $(X_3 : Y_3 : Z_3)$
17: **else**
18: **return** ∞
19: **end if**
20: **end if**
21: $Z_3 \leftarrow Z_1 \cdot T_1$
22: $T_3 \leftarrow T_1^2$
23: $T_4 \leftarrow T_3 \cdot T_1$
24: $Y_3 \leftarrow T_3 \cdot X_1$
25: $T_1 \leftarrow 2 * Y_3$
26: $X_3 \leftarrow T_2^2$
27: $X_3 \leftarrow X_3 - T_1$
28: $X_3 \leftarrow X_3 + T_4$
29: $Y_3 \leftarrow Y_3 - X_3$
30: $T_3 \leftarrow Y_3 \cdot T_2$
31: $T_1 \leftarrow T_4 \cdot Y_1$
32: $Y_3 \leftarrow T_3 - T_1$
33: **return** $(X_3 : Y_3 : Z_3)$

4.2.3 Scalar Point Multiplication

With the left-to-right binary method the point doubling in mixed Jacobian affine coordinates and the point doubling for Jacobian coordinates can be used to calculate $k \cdot P$, where k represents a 160-bit integer and P represents a point on the elliptic curve. With k and P as input the algorithm will output $k \cdot P$.

Algorithm 3 Left-to-right binary method for point multiplication

Require: $k = (K_{t-1}, \ldots, k_1, k_0)_2$, $P \in E(\mathbb{F}_p)$
Ensure: $k \cdot P$
1: $Q \leftarrow \infty$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)

4.3 Primefield Arithmetic Optimization

When choosing an algorithm for the 160-bit multiplication it is important to consider the hardwares characteristic, such as processor word-size and number of general purpose registers. Since multiplication and SRAM access are the most expensive operations on the ATmega128L basically two different optimizations strategies are possible: reduce the number of multiplication or reduce SRAM usage. The first approach would be to implement Karatzuba [MvOV97] and the second some kind of *improved schoolbook algorithm*. The *hybrid multiplication* [GPW$^+$04] is a memory optimized variant of the *schoolbook algorithm*. A special characteristic of the algorithm is that the computational cost rises linearly with smaller numbers of registers and processor word size. It is much easier to implement than Karatzuba and hence much easier to port to different platforms. For these reasons the *hybrid multiplication* was chosen.

The next two following algorithms are the schoolbook variants *operand scanning multiplication* also known as *column wise multiplication* and *product scanning multiplication* also known as the *row wise multiplication*. These two nested into each other result in *hybrid multiplication*.

4.3.1 Operand scanning multiplication

The operand scanning multiplication can be efficiently implemented with the following algorithm.

Algorithm 4 Operand Scanning Multiplication

Require: Integers $a, b \in [0, p-1]$
Ensure: $c = a \cdot b$
1: $c[i] \leftarrow 0$ for $0 \leq i \leq t-1$
2: **for** i from 0 to $t-1$ **do**
3: $U \leftarrow 0$
4: **for** j from 0 to $t-1$ **do**
5: $(UV) \leftarrow c[i+j] + a[i] \cdot b[j] + U$
6: $c[i+j] \leftarrow V$
7: **end for**
8: $c[i+j] \leftarrow U$
9: **end for**
10: **return** (c)

4.3.2 Product scanning multiplication

The product scanning multiplication can be efficiently implemented with the following algorithm.

Algorithm 5 Product Scanning Multiplication

Require: Integers $a, b \in [0, p-1]$
Ensure: $c = a \cdot b$
1: $r_0 \leftarrow 0$, $r_1 \leftarrow 0$, $r_2 \leftarrow 0$
2: **for** k from 0 to $2t-2$ **do**
3:     **for** each element of $\{(i,j) | i+j=k, 0 \le i,j \le t-1\}$ **do**
4:         $(UV) \leftarrow a[i] \cdot b[j]$
5:         $(\epsilon, r_0) \leftarrow r_0 + V$
6:         $(\epsilon, r_1) \leftarrow r_1 + U + \epsilon$
7:         $r_2 \leftarrow r_2 + \epsilon$
8:     **end for**
9:     $c[k] \leftarrow r_0$, $r_0 \leftarrow r_1$, $r_1 \leftarrow r_2$, $r_2 \leftarrow 0$
10: **end for**
11: $c[2t-1] \leftarrow r_0$
12: **return** (c)

4.3.3 Hybrid Multiplication

The *hybrid multiplication* combines the *product scanning* method with the *operand scanning* method. The two outer nested loops perform *column wise multiplication* and the two inner nested loops describe *row wise multiplication*. The number of rows calculated to perform a column are called *column width* d[1].

The *hybrid multiplication* uses several parameters that adapt the algorithm to the hardware.

$$d = \max\{i \mid 1 \le i \ge n, r \ge 3i + \lceil \log_2 (n/i)/k \rceil\} \tag{4.1}$$

where k denotes the bit length of one operand word and fittingly set to the CPU word size, which is 8-bit. Whereas, m denotes the bit length of the operands. In this case the operands have a length of 160 bit, therefore the resulting operand size is $n = 20$. Assuming the parameters as constant for this implementation the SRAM usage scales with the number of registers r that the algorithm shall use. The concerning parameter is called *column width* (d) and equals the number of partial products per row.

[1] Other bottlenecks also require some spare registers as will be shown later on.

Algorithm 6 describes the *hybrid multiplication*. The two operands that are multiplied are displayed as *mem_a* and *mem_b* and are temporarily loaded from SRAM into registers $a_{d-1}, \ldots, a_0$ and b. The result is accumulated in registers $r_{2d-1+\lceil log2(n/d)/k \rceil}, \cdots$, r_0. The lower d registers are stored to *mem_c* at the end of each column. The variables indicated by *mem* stay in the SRAM during the whole multiplication.

Algorithm 6 Hybrid multiplication

Require: n : operand size in words,
d : column width,
$mem_a[\lceil n/d \rceil \cdot d-1 \dots 0]$: multiplicand A,
$mem_b[\lceil n/d \rceil \cdot d-1 \dots 0]$: multiplier B
Ensure: $mem_c[\lceil n/d \rceil \cdot 2d-1 \dots 0]$: result $C = A \cdot B$
1: **for** i from 0 to $\lceil n/d \rceil - 1$ **do**
2: **for** j from 0 to i **do**
3: $(a_{d-1}, \dots, a_0) = mem_a[(i-j+1) \cdot d - 1 \dots (i-j) \cdot d]$
4: **for** s from 0 to $d-1$ **do**
5: $b = mem_b[j \cdot d + s]$
6: **for** t from 0 to $d-1$ **do**
7: $(r_{2d-1+\lceil log2(n/d)/k \rceil}, \dots, r_0) = (r_{2d-1+\lceil log2(n/d)/k \rceil}, \dots, r_0) + a_t \cdot b \cdot 2^{k \cdot (t+s)}$
8: **end for**
9: **end for**
10: **end for**
11: $mem_c[(i+1) \cdot d..i \cdot d] = (r_{d-1}, \dots, r_0)$
12: $(r_{d-1+\lceil log2(n/d)/k \rceil}, \dots, r_0) = (r_{2d-1+\lceil log2(n/d)/k \rceil}, \dots, r_d)$
13: $(r_{2d-1+\lceil log2(n/d)/k \rceil}, \dots, r_d) = 0$
14: **end for**
15: **for** i from $\lceil n/d \rceil$ to $2 \cdot \lceil n/d \rceil - 2$ **do**
16: **for** j from $i - \lceil n/d \rceil + 1$ to $\lceil n/d \rceil - 1$ **do**
17: $(a_{d-1}, \dots, a_0) = mem_a[(i-j+1) \cdot d - 1 \dots (i-j) \cdot d]$
18: **for** s from 0 to $d-1$ **do**
19: $b = mem_b[j \cdot d + s]$
20: **for** t from 0 to $d-1$ **do**
21: $(r_{2d-1+\lceil log2(n/d)/k \rceil}, \dots, r_0) = (r_{2d-1+\lceil log2(n/d)/k \rceil}, \dots, r_0) + a_t \cdot b \cdot 2^{k \cdot (t+s)}$
22: **end for**
23: $mem_c[(i+1) \cdot d \dots i \cdot d] = (r_{d-1}, \dots, r_0)$
24: $(r_{d-1+\lceil log2(n/d)/k \rceil}, \dots, r_0) = (r_{2d-1+\lceil log2(n/d)/k \rceil}, \dots, r_d)$
25: $(r_{2d-1+\lceil log2(n/d)/k \rceil}, \dots, r_d) = 0$
26: **end for**
27: **end for**
28: $mem_c[(i+1) \cdot d \dots i \cdot d] = (r_{d-1}, \dots, r_0)$
29: **end for**
30: **return** (mem_c)

4.3.4 Inversion

The inversion can be efficiently implemented with the following algorithm, from [HMV04]. The algorithm was slightly changed: the modular bisection, e.g. $x_1 \leftarrow x_1/2 \bmod p$ was originally solved within the algorithm. Since we already implemented an efficient bisection modulo p we outsourced this. Therefore Algorithm 8 can be used for all bisections required in this algorithm. Note that Algorithm 8 can in this case also be used for the non-modular bisection, e.g. $u \leftarrow u/2$, because it is only applied to even integers.

Algorithm 7 Binary algorithm for inversion in $\mathbb{F}_p$

Require: Prime p and $a \in [1, p-1]$
Ensure: $a^{-1} \bmod p$

```
1:  u ← a, v ← p
2:  x1 ← 1, x2 ← 0
3:  while (u ≠ 1 and v ≠ 1) do
4:     while u is even do
5:        u ← u/2
6:        x1 ← x1/2 mod p
7:     end while
8:     while v is even do
9:        v ← v/2                 (Algorithm 8)
10:       x2 ← x2/2 mod p         (Algorithm 8)
11:    end while
12:    if u ≥ v then
13:       u ← u − v
14:       x1 ← x1 − x2
15:    else
16:       v ← v − u
17:       x2 ← x2 − x1
18:    end if
19: end while
20: if u = 1 then
21:    return (x1 mod p)
22: else
23:    return (x2 mod p)
24: end if
```

4.3.5 Reduction

The prime field reduction of SEC2 curves can be done in the same way as for NIST curves. The best performing reduction in this case is based on the fact that Mersenne Primes are of the form $2^n - 1$. In binary representation they consist almost exclusively of 1s. The here used Mersenne Prime is according to the chosen curve ($p = 2^{160} - 2^{31} - 1$).

To figure out how to reduce a value bigger than p we can stick to the fact, that $2^{160} - 2^{31} - 1 \equiv 0 \pmod{p}$ and therefore $2^{160} \equiv 2^{31} + 1 \pmod{p}$.

Furthermore any element of the according prime field can be displayed as a sum of the type

$$A = a_0 * 2^0 + a_1 * 2^8 + ... + a_{18} * 2^{144} + a_{19} * 2^{152} \mod p \tag{4.2}$$

where a_i represents one 8-bit word.

The result of a 160-bit long term multiplication can be displayed as

$$A = a_0 * 2^0 + a_1 * 2^8 + ... + a_{38} * 2^{304} + a_{39} * 2^{312}.$$

Now every byte can be reduced separately. For example:

$$a_{20} * 2^{160} \equiv a_{20} * 2^{31} + a_{20} * 2^0 \mod p$$

4.3.6 Bisection

The modular bisection can be done as follows: if a is even then $a \leftarrow a/2$ else $a \leftarrow (a+p)/2$. An efficient implementation should use a shift to realize the bisection. Algorithm 8 is designed in a way that allows to realize every bisection by executing shifts. Furthermore the algorithm will also work for for non-modular bisections of even integers. It is easier to check for a being even after the shift is done, because the bit that is shifted out is stored in the carry bit, which can be easily checked. Therefore we created the following algorithm. Note that $\lfloor a/2 \rfloor$ represents the result of shifted integer.

Algorithm 8 Bisection with shift in $\mathbb{F}_p$

1: $a \leftarrow \lfloor a/2 \rfloor$
2: **if** a even **then**
3: $\quad a \leftarrow a + \lceil p/2 \rceil$
4: **end if**
5: **return** a

5 Implementation of SECP160r1 on ATmega128L

This section starts with a description of the road map for the implementation. Subsequently, the focus is put on the multiplication, since this is the most important part, respectively the part with the most potential for optimization. More than 77% of the process time of an operation on the curve is spent with multiplication. Two approaches were necessary to figure out bottlenecks and solutions. The technical experience from the first approach was applied in the second approach, which is exposed in detail. Afterwards the other prime field operations are described. Finally the curve arithmetics implementation is explained.

The prime field implementation has to provide 160-bit arithmetic for multiplication, addition, subtraction and bisection. To adapt the algorithms in the best possible way to the hardware they are realized using assembly. As mentioned before the reduction can be implemented very efficiently due to the chosen Mersenne prime. Addition and subtraction can be done without special optimization, because they can be realized by concatenated additions and subtraction with carry. The highest computational cost lies in the 160-bit multiplication of the prime field.

5.1 Multiplication

The multiplication is realized as plain 160-bit long number arithmetic. The resulting 320-bit value is subsequently reduced thus resulting in a prime field multiplication. The multiplication is done by using an optimized variant of the *schoolbook algorithm*.

This kind of multiplication technique can be divided into two types: the operand scanning and the product scanning multiplication. Algorithms for both are introduced in Section 4.3.1, and Section 4.3.2. The section starts with a general effort analysis of the *schoolbook multiplication*, resulting in a theoretical minimum effort.

Afterwards bottlenecks will be shown, analyzed, and explained in detail together with the algorithm dealing with this bottleneck. Finally the two implementation approaches are described, whereas the first is focusing on one bottleneck, while the second deals with both bottlenecks.

5.1.1 Schoolbook Multiplication

There exist various variants of the *schoolbook algorithm.* Generally the multiplication is divided in several small multiplications that are accumulated to get the final result. These core operations will be referred to as the *elementary instruction block* (EIB), see Figure 5.1.

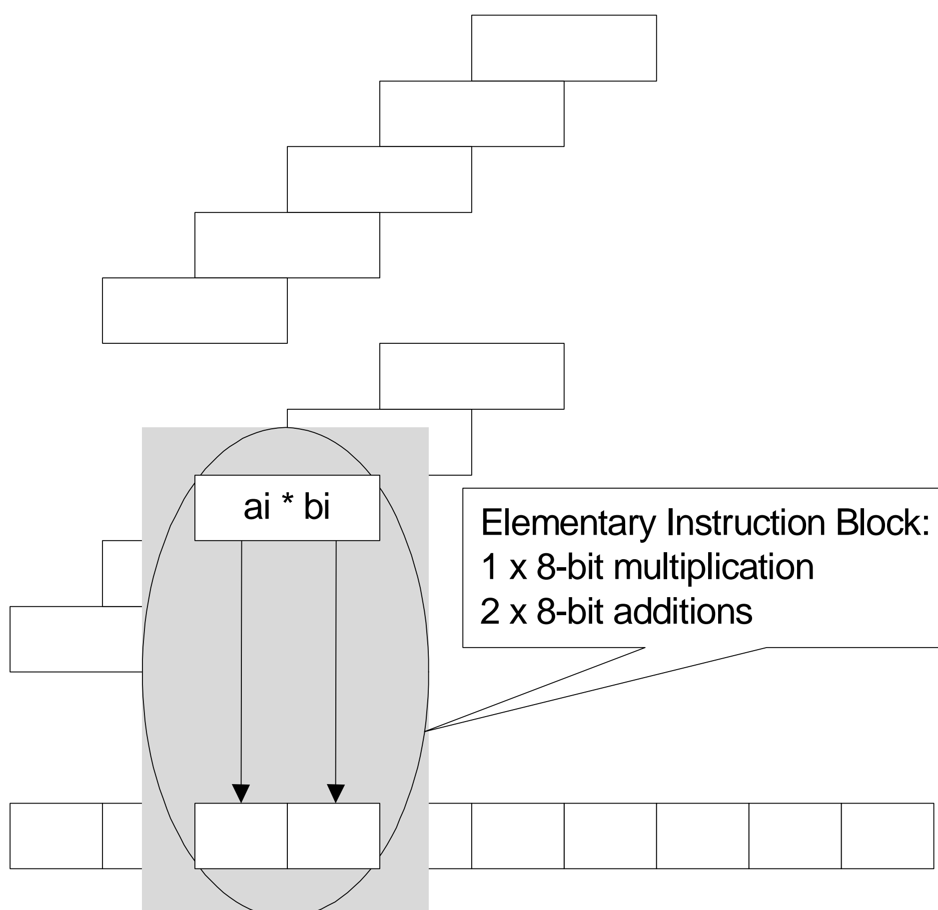

Figure 5.1: Elementary Instruction Block (EIB)

The Figure shows an 8-bit multiplication resulting in two 8-bit words. Each of the two product-words has to be accumulated by one 8-bit addition.

Regardless of optimization, in general two approaches are possible to sort the sequence of execution of these blocks: scanning the operants from left-to-right or right-to-left[1] to select EIBs is called row wise multiplication, see Figure 5.2. A second way to sort the summands before they are accumulated is scanning the product, which means sorting the blocks by bit length. Mixing both methods up, where the latter mentioned technique is nested in the former one, is called *hybrid multiplication*, [GPW$^+$04]. By using the *hybrid multiplication* method the overhead caused by SRAM usage can be minimized. As the inner nested method will handle most of the carry bits it is crucial for the speed.

To make this clear, one *elementary instruction block* defines one 8-bit multiplication and the addition of the 16-bit product to the accumulator, which needs two 8-bit addition. It neither states anything about the order in which the multiplications and additions are done, nor about the handling of carry bits, if any, nor anything about how operands are fetched from SRAM or the accumulator written back to SRAM. As long as a schoolbook strategy is implemented, a 160-bit multiplication on an 8-bit CPU will have to be performed by 400 *elementary instruction blocks*. The type of *schoolbook algorithm* will define a general strategy on how to fetch operands and may as well handle carry bits, avoid carry bits, or none of both. In fact an efficient handling of carry bits might strongly depend on the hardware. Due to this the final effort of carry handling is dependent on the CPU, as the next sections will describe in detail.

5.1.2 Theoretical Minimum Effort for Modular Multiplication

For a 160-bit multiplication 400 *elementary instruction blocks* have to be executed, where one EIB (one multiplication and two additions) equals four cycles on the ATmega128L. From this arises a minimum effort of 400 8-bit multiplications and 800 additions, resulting in 1600 cycles as theoretical minimum. Of course in practice the operands have to be loaded to the registers first and pointers have to be handled. Other overhead like carry bit handling etc. also occur. As some problems might need a lot registers sometimes it is worth to buffer registers to the SRAM.

[1] This is what is taught in school when learning the multiplication the first time - probably giving the algorithm its name

A small example for overhead caused by loops: loops, if any, can produce a big overhead as well. For example if a loop is needed to count to a value bigger than 255 two registers are needed on an 8-bit CPU. By this just for increasing or decreasing the counter two additions are needed. One for the real addition and one to add the carry to the next higher significant part of the counter. It is faster and easier to do this addition, as checking if an addition is needed takes the same time as the addition itself. Again a bigger overhead occurs for checking the counter against specific values if it has to be held in two registers instead of one. However even a loop with a 1-word-counter needs one addition and one jump (including the break condition). If only very few instructions are executed by the loop the overhead is immense. If the loop executes instructions with four clock cycles in total the overhead strikes with 50%. This overhead doubles for loops with a 2-word-counter.

5.1.3 Bottlenecks

The variants of the schoolbook multiplication method differ in the sequence of the core multiplications and additions. The EIBs can be executed in any thinkable order. Two bottlenecks occur here: since the operands are 160-bit in length they do not fit both in the 32 8-bit registers of the micro controller. Reloading parts of the operands multiple time produces a large overhead by handling pointers and executing SRAM reading operations. Secondly, the accumulation of the blocks has to be done in an order that does not produce a large overhead by handling carry bits. In practice this can be quite challenging, as will be shown in the following.

The effort of SRAM usage scales between 120 and 840 operations or 240 and 1680 clock cycles plus a small overhead due to pointer handling. An effort of 1680 clock cycles means loading and storing operand and results would take more time than the multiplication itself! 240 clock cycles on the other hand would still increase the minimum effort by 15%, but is much more acceptable than an increase of about 100%. Details about SRAM usage are decribed in combination with the *hybrid multiplication* later on in this chapter, because this multiplication strategy is used to scale the SRAM usage.

The effort to handle carry bits depends much on the abilities of the used hardware and the used schoolbook multiplication variant. An operand scanning strategy without further optimization on the ATmega128L would take two adds and one move instruction per *elementary instruction block*. That are 1200 instructions in total. If used in a *hybrid multiplication* a few additional carry bits will appear for the product scanning algorithm. This represents an increase of the minimum effort by at least 75%. Hardware specific solutions have to be found here. The inner nested row

wise multiplication of the *hybrid multiplication* deals with most of the carry bits, therefore details about occurring carry bits are given there.

Since the overhead caused by handling carry bits cannot be calculated in advance or sufficiently by algorithms, this problem will be solved during the implementation process, contrary to the effort of SRAM usage. The overhead of the SRAM usage caused by pointers can be neglected.

A small overhead by other operations like initialization or jumps remains without being explained in detail and is neglected as well.

In fact this is based on the fact that the *elementary instruction blocks* are not executed in a loop. Not assuming this would mean at least three cycles overhead per block resulting in 1200 additional cycles or an overhead of 75%, respectively. Therefore all implementations are started with heavy macro usage instead of loops. As soon as a sufficient multiplication strategy is found code size reduction can be performed. How this could be done is explained in Section 5.3.4.

5.1.4 Operand Scanning Strategy and Carry Handling

Figure 5.2 illustrates the general work flow of a product scanning algorithm and the arrangement of EIB multiplications. It is an example with two 5-word operands. Two mechanisms can be found here that are nested. The inner mechanism multiplies each word of one operand with one word of the other operand. The partial products are accumulated. The sum is equivalent to the multiplication of an operands' word with the whole other operand: a multiplication of one n-word multiplicand with one word of the other multiplicand. In the ideal case *add with carry* can be used to accumulate all the products. Unfortunately, the additions of two blocks overlap each other as can be seen in Figure 5.3.

Algorithm 4 presented in Section 4.3.1 has an integrated carry handling. The algorithm is accumulating the summands in a 2-word accumulator called UV in a way that UV will always stay in its range and thus no carry handling is required - in theory. The algorithm needs to perform two additions and one MOVE operation in the inner part and another MOVE operation in the outer part. UV has the size of two words, which means adding a value of one word in size would recommend the hardware to perform one 2-word addition. Usually one would like to chose the word size of the algorithm according to the word size of the CPU. This way the multiplication would produce on an ATmega128L a 16-bit value that could be used as UV, followed by two additions. Adding one 8-bit value to UV needs one 16-bit addition

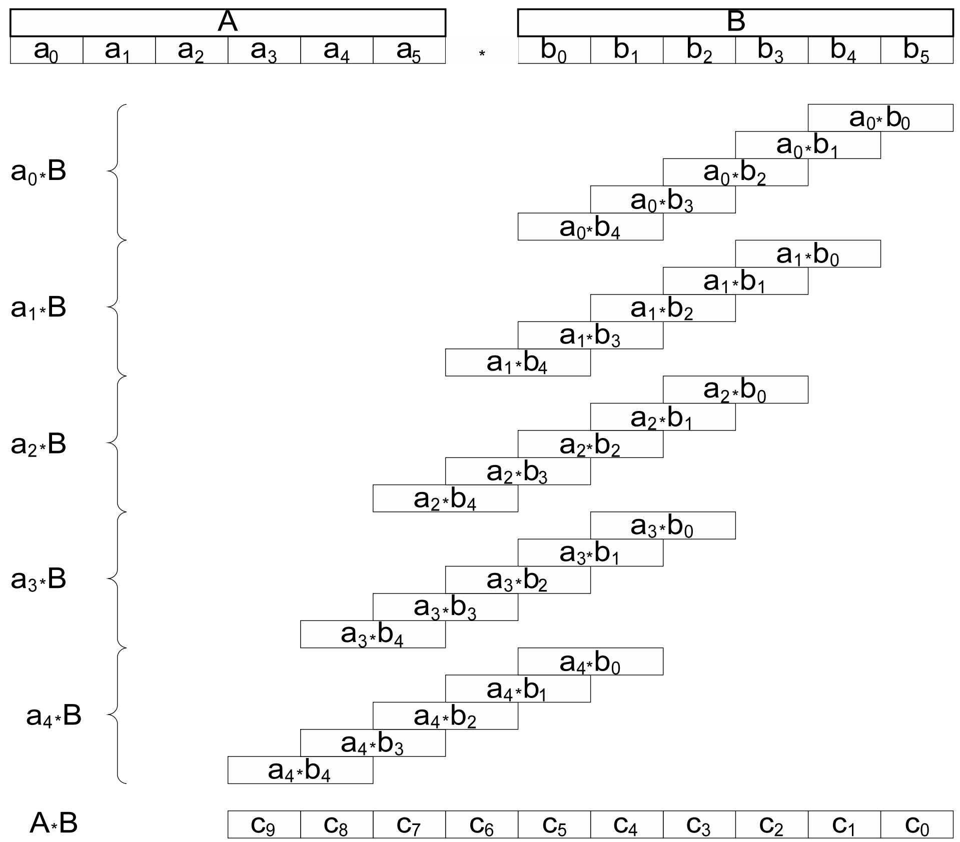

Figure 5.2: Row Wise Multiplication

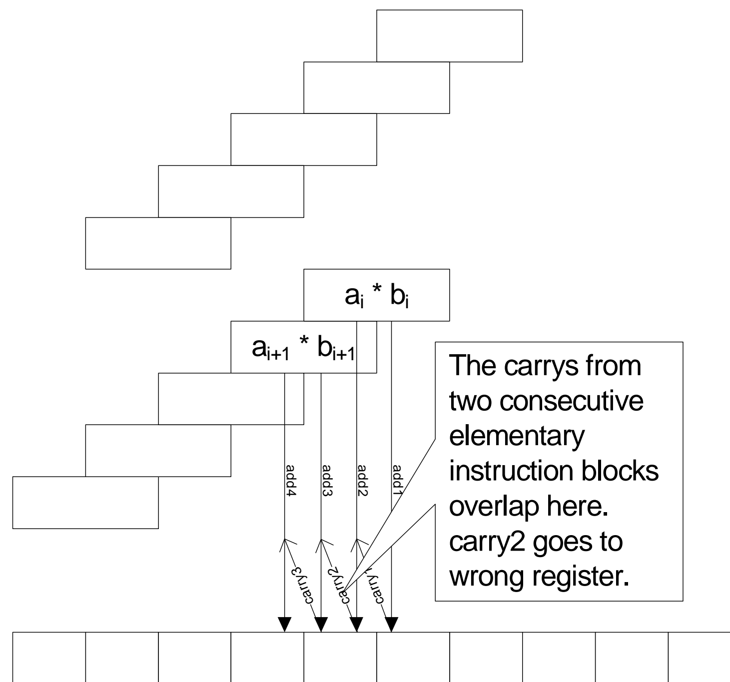

Figure 5.3: Elementary Instruction Blocks Overlap Each Other

or one 8-bit addition to V followed by an operation that adds any occurring carry bit to U. As most CPUs handling words of the size x, cannot perform additions of the size $2*x$, the effort with this algorithm is more like four additions plus one MOVE in the inner part and one MOVE in the outer part.

Other algorithms implementing the operand scanning techniques show similar disadvantages. The algorithm has to be adapted to the hardware, to minimize the overhead by carry bits. On the other hand the efficient SRAM usage of the *hybrid multiplication* is narrowed by the available registers. Unfortunately, the effort for handling carry bits cannot be calculated in advance contrarily to the effort produced by the SRAM usage using a *hybrid multiplication*. The hardware allows different methods to add, store and restore one or multiple carry bits. For this reason it is most difficult for an algorithm to present efficient carry handling. The possible solutions are the results of various experiments.

Two approaches were done, the first focused on the SRAM usage serving the *hybrid multiplication* with all possible registers, using the remaining registers for carry handling. The second approach was done with the next smaller possible amount of reserved registers, allowing a more efficient working carry handling.

5.1.5 Hybrid Multiplication and SRAM Access

Another general approach to sort *elementary instruction blocks* is to scan the product and execute all those blocks which result bit lengths match the current position of the product. This is called *row wise multiplication* or *product scanning multiplication*, see Figure 5.4. The picture shows that the number of summands per row is at most n, where n is the number of words per operands. For $n > 2$ more than one carry bit may be caused by the additions that has to be added to the next column. This makes the use of *add with carry* nearly impossible. The method is yet quite useful as it will be implemented as the outer part of the *hybrid multiplication*. It will not execute *elementary instruction blocks* but rather call the *row wise multiplication* to do this and will then in turn accumulate the results of the latter, as illustrated in Figure 5.5. The effort for buffering row-carry bits is therefore low compared with the effort of handling column-carry bits. However, the *product scanning multiplication* combined with the *operand scanning multiplication* is called *hybrid multiplication* and significantly reduces the SRAM usage.

In Table 5.1 all possible column width [2] for the ATmega128L micro controller

[2] $d = \max\{i \mid 1 \leq i \geq n, r \geq 3i + \lceil \log_2 (n/i)/k \rceil\}$

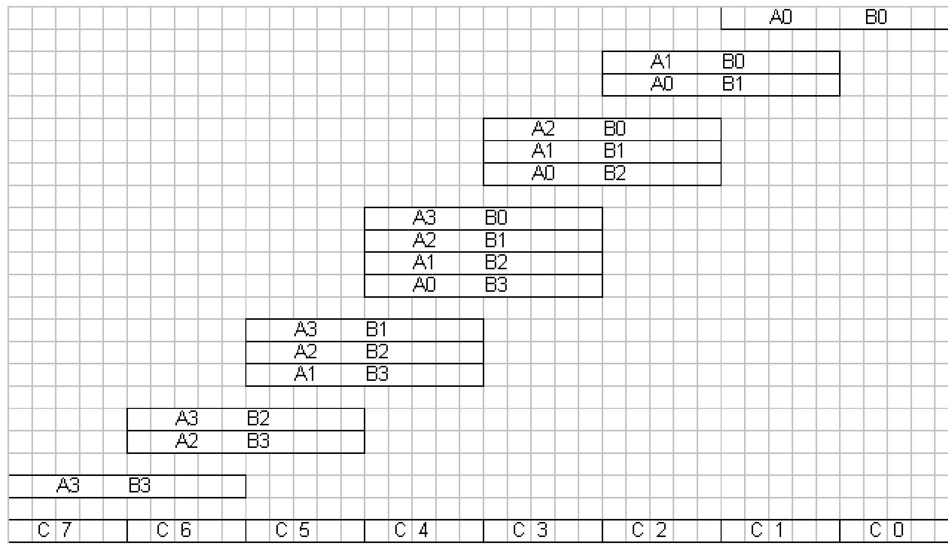

Figure 5.4: Column Wise Multiplication

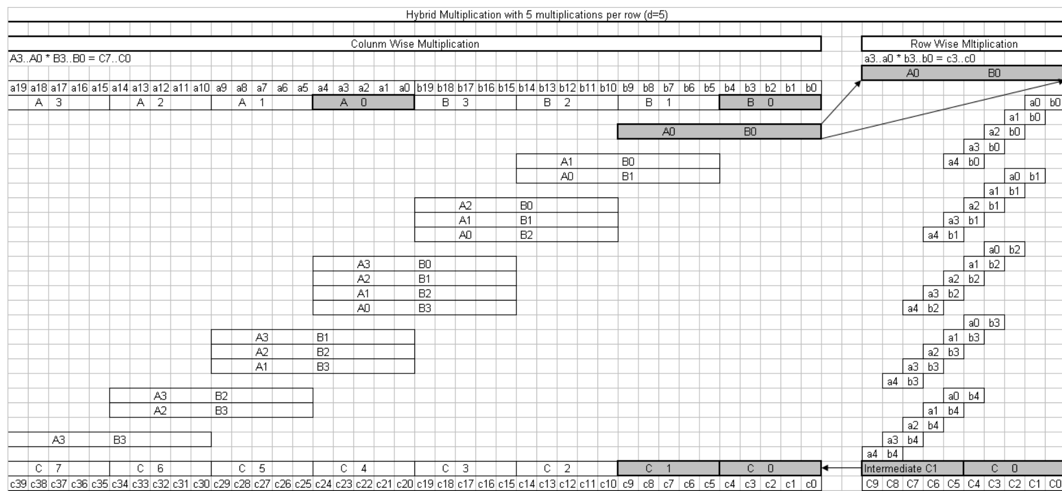

Figure 5.5: 160-bit Hybrid Multiplication on ATmega128L with $d = 5$

Column Width	**Register usage #of registers required**			**SRAM usage # of load and store**		
d	**row**	**column**	**hybrid**	**row**	**column**	**hybrid**
2	22	5	8	460	840	440
3	22	5	11	460	840	308
4	22	5	14	460	840	240
5	**22**	**5**	**17**	**460**	**840**	**200**
6	22	5	20	460	840	174
7	22	5	23	460	840	156
8	22	5	26	460	840	140
9	22	5	29	460	840	130
10	**22**	**5**	**32**	**460**	**840**	**120**

Table 5.1: SRAM and Register Requirements of the Hybrid Multiplication

are listed. The register usage describes the amount of registers needed for storing operand and accumulator. The SRAM usage represents the amount of SRAM load and store operations, which bring parts of the operands to the registers and store the result. It is clearly visible that the column wise multiplication is the most inefficient variant concerning SRAM operations, while it has the lowest register consumption at all. The row wise variant uses far more registers than the column wise strategy, but also has a much better SRAM efficiency. It should be mentioned that there are 20 registers required for holding the whole accumulator. If the accumulator would not fit into the registers the effort of this algorithm would significantly grow. The third displayed strategy is the *hybrid multiplication* using the *column wise multiplication* nested in the *row wise multiplication*. It lowers the SRAM usage with increasing number of used registers. This is because the efficient SRAM usage arises from the multiple usage of operand pieces, which have to be held in the registers. Remarkable is that the *hybrid multiplication* obviously outperforms the other two strategies for most column widths.

The two bold lines for $d = 5$ and $d = 10$ are those implemented. The latter is optimized for SRAM access while the former focuses on reducing the overhead produced by handling carry bits. Both approaches are described in the following. For comparison reasons *row wise* and *column wise multiplication* are illustrated as well, even though they do not depend on colunm width.

The optimal column width depends on available registers r, the operand size n, and the bit length k. d should be chosen as large as possible, because the SRAM usage lowers with bigger d

5.2 Modular Multiplication with Column Width 10

This approach will focus on a maximum saving in SRAM usage to reduce execution time. As a matter of fact this leads to massive register usage of the *hybrid multiplication* algorithm. A column width of $d = 10$ is selected using nearly all available registers. The main bottlenecks are assumed to be first the effort for SRAM usage and second the overheads for handling carry bits. Since the effort for theses bottlenecks is going to be measured the code was completely unrolled by usage of macros to reduce other overhead as far as possible. On the one hand this increased the code size enormously but on the other hand this makes extensive changes in the algorithms easier. When a sufficient solution for the carry handling is found the next step could be to reduce the code size without producing a large overhead in runtime by applying a loop.

5.2.1 Adapting the Row Wise Multiplication to the Target Platform

The *hybrid multiplication* with a column width of $d = 10$ will need 120 SRAM operations respectively 240 clock cycles plus a neglected overhead caused by pointer handling. The *hybrid multiplications* inner nested algorithm is the operand scanning multiplication. According to algorithm 5 it will execute the *elementary instruction blocks* as follows:

```
1  (UV)    <- c[i+j] + a[i]*b[j] + U
2  c[i+j] <- U
```

Taking 8-bit as word-size $a[i] * b[j]$ represent an 8-bit multiplication. U and $c[i + j]$ are 8-bit words. (UV) is a 16 bit word and the two additions needed to calculate it cannot exceed the value $0xFFFF$. As an 8-bit multiplications' maximum value is $0xFE01$ and the two words added to it are at maximum $0xFF$ this results in a total maximum of $0xFFFF$. Unfortunately the two additions are 16-bit addition, which most 8-bit micro controllers cannot handle. A simple 8-bit ADD to the low significant byte V may result in a carry addition to the high significant byte U. Hence, the algorithm must be adapted to the hardware. Furthermore, (UV) is a buffer requiring two registers, that are rare because nearly all registers are required to hold the operands and the accumulator. Figure 5.6 shows to the best knowledge the optimum realization.

The resulting avr-pseudo-code looks like:

```
1 mul ai, bi
2 add R0, RU
3 adc R1, zero
4 add c[i+j], R0
5 adc R1, zero
6 mov RU, R1
```

The 8-bit multiplication always puts the results in the registers R0 and R1 of the micro controller. The 16-bit word (UV) is represented by two 8-bit registers. The most significant byte of (UV) will be calculated in R0 and moved to RU at the end, for usage in the next iteration. The least significant byte of (UV) is calculated in $c[i+j]$, so it has not to be moved to this place at the end. In detail the row wise multiplication consists of the following six code lines:

1. The code starts with the 8-bit multiplication of a_i and b_i and the 16-bit result appears in R0 and R1. (line 1)

2. RU represents u and is added to $R0$. As $R0$ will later on be added to $c[i+j]$ the accumulation is done to the low significant byte of (UV). (line 2)

3. This addition must be followed by an addition of a possibly occurring carry bit to the most significant byte of (UV). (line3)

4. Next is the addition of $R0$ to the least significant byte of (UV), which is $c[i+j]$. (line 4)

5. Again an occurring carry bit has to be added to the most significant byte of (UV). (line 5)

6. Finally the most significant byte of (UV) hast to be buffered to RU, for usage in the next iteration. (line 6)

The above described application of the algorithm needs three clock cycles per *elementary instruction block* to handle carry bits. As 400 blocks have to be executed this represents an overhead of 1200 cycles. This would mean an overhead of 65% relating to the sum of the minimum effort and the SRAM effort of in total 1840 clock cycles. A saving of a single clock cycle will reduce the overhead to 43% or 800 clock cylcles, respectively. The following Section shows how this could be achieved by modifiying the operand scanning algorithm.

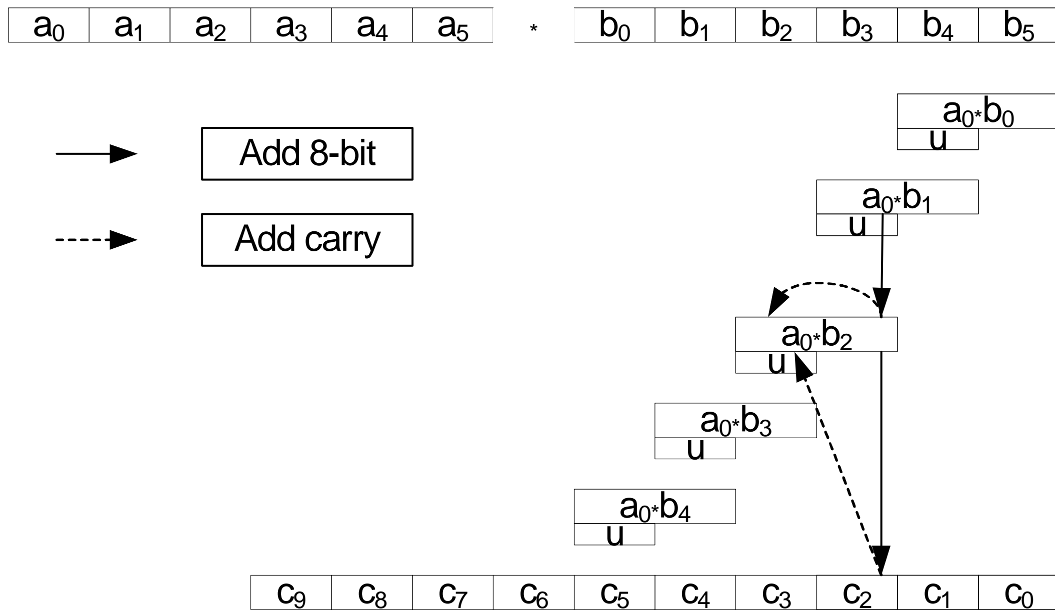

Figure 5.6: UV Carry Handling

5.2.2 Secure Carry Add

This approach will not be based on a 2-word-sized buffer. Furthermore it will not serve carry-free multiplication, but rather works with 8-bit words and handles carry bits. To the best of our knowledge this technique has not yet been published. When executing an 8-bit multiplication the maximum result is $0xFE01$. Therefore the least significant byte varies in the range of $0x00$ to $0xFF$ while the most significant byte stays in the range between $0x00$ and $0xFE$, as shown in Figure 5.7. The white fields mark words on which a carry bit can by added securely, while gray shaded words may not be accumulated with a carry bit without the possibility of a newly occurring carry bit. The improved column wise multiplication will use this feature to handle occurring carry bits, as shown in Figure 5.8. The runtime of this algorithm depends strongly on the device's ability of buffering carry bits. Usually a carry bit can be copied to a register, either alone or by copying the whole CPU flag register. Alternatively, ba adding $0x00$ to the buffer with the ADD WITH CARRY instruction, it is possible to save the carry bit. Another possibility is a *rotate trough carry* instruction. These different possibilities of applying the algorithm to the hardware differ both in runtime and register usage.

Unfortunately no combination could be found to buffer a carry for adding it in

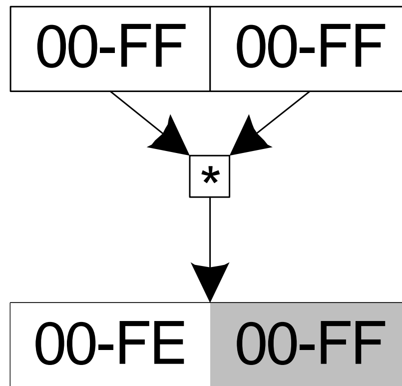

Figure 5.7: Save Carry Add

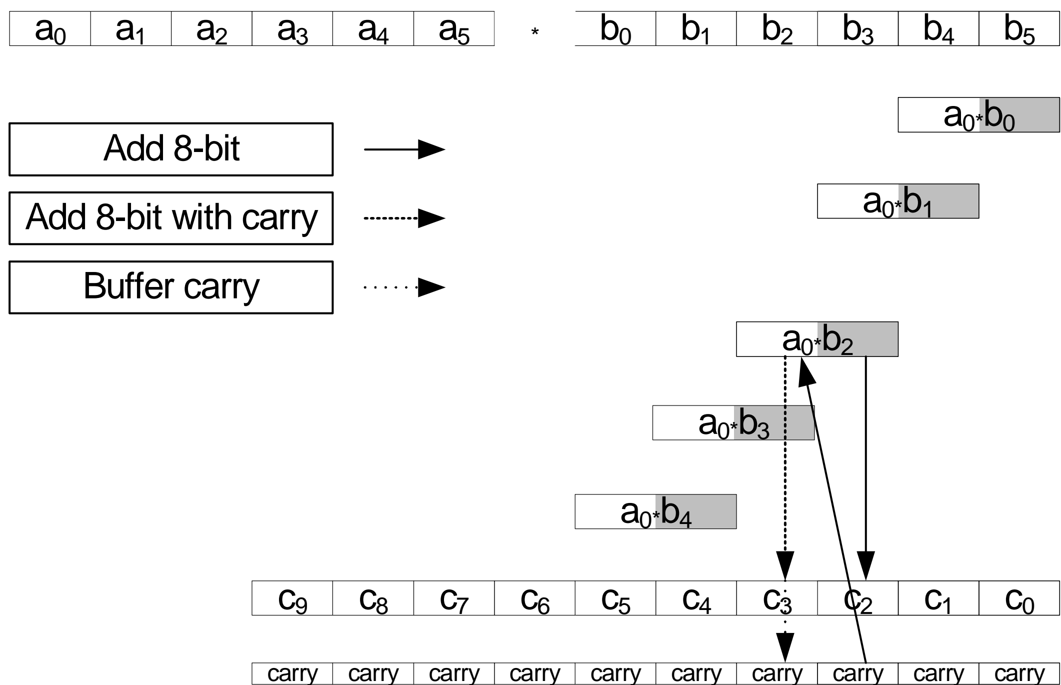

Figure 5.8: Carry Addition Scheme of an Unrolled Column

the following iterations that would be executable multiple times. When adding or rotating the carry bit to a buffer, the buffer would increase to a value of more than one in the next iteration. Therefore the buffer has to be cleared after the carry has been used. The clearing is increasing the overhead again to three clock cycles per *elementary instruction block*. Several variants for secure carry add follow, but all do need three cycles to handle the carry bit.

ADC Variant

```
1  mul ai, bj
2  add R1, cbuff        //secure carry add
3  eor cbuff, cbuff     //clear cbuff
4  add ci, R0
5  adc ci+1, R1
6  adc cbuff, zero     //add carry to cbuff
```

This approach requires two additional registers: CBUFF, to buffer the carry and zero, which has to be initialized to 0x00 and will never be set to a different value. It is required to add the carry bit to CBUFF.

ROL Variant

```
1  mul ai, bj
2  add R1, cbuff        //secure carry add
3  eor cbuff, cbuff     //clear cbuff
4  add ci, R0
5  adc ci+1, R1
6  rol cbuff            //rotate carry to bit 0 of cbuff
```

This approach only needs one additional register, the carry buffer cbuff.

IN Variant

```
1  mul @3, @4
2  ANDI @2, 0b00000001  //mask the carry bit
3  add R1, @2           //secure carry add
4  add @0, R0
5  adc @1, R1
6  in  @2, 0x3F         //copy the hole CPU flagbyte to cbuff
```

This approach also only requires one additional register. This is the variant that is implemented, because it needs one register less that variant one. The second

variant would also have been possible. As a matter of fact it has the same running time as algorithm 4, which hardware adaption is described in Section 5.2.1. The total runtime of this approach ended up in about 4500 clock cycles for a 160-bit multiplication. A detailed list of the used instructions is presented in Table 5.2.

mnemonic	**# used**	**#CC/instr.**	**clock cycles**	**min effort**	**carry**
mul	400	2	800	800	
add	1360	1	1360	800	400
in	400	1	400	0	400
andi	400	1	400	0	400
ld	88	2	176	0	160
st	40	2	80	0	80
other	528	1	528	0	
SUM			**3744**	**1600**	**1440**

Table 5.2: Required Instructions and Clock Cycles for a 160-bit Multiplication with $d = 10$

Table 5.2 describes the required instruction the *hybrid multiplication* with a column width of $d = 10$. Furthermore the table shows the required clock for the *minimum effort* and the required clockcycles for handling carry bits.

5.2.3 Use Add With Carry Instruction

Another approach to reduce the needed clock cycles for carry handling is based on a different sequence of executing the *elementary instruction block* in the inner part of the operand scanning algorithm. The idea is to eliminate the overlap of the instruction blocks, see Figure 5.9. The inner part of the column wise multiplication is split into two parts. The first parts picks those blocks whose additions to the accumulator can be executed consecutively. The remaining blocks have the same feature and are executed in the second part. This way the instruction *add with carry* can always be used. To the best of our knowledge this method has not yet been published. In theory this is slightly increasing the overhead of the pointer handling but reducing the carry handling to zero!

When applying this method to the ATmega128L the carry flag is unfortunately overwritten by other instructions executed in the mean time between two iterations. But this method still scales well, because now a two clock cycle approach for handling the carry is possible. On different hardware the carry flag may stay untouched

bringing this strategy to its maximum efficiency. The pseudo code shows an example realizing this strategy:

```
1  mul @3, @4
2  rol cbuff        // restore carry bit
3  adc R1, @2
4  adc @0, R0
5  ror cbuff        // store carry bit
```

After the multiplication is done, the carry bit from the last iteration is recovered to bit seven of cbuff. The two *additions with carry* follow. An occurring carry bit is stored to bit seven of cbuff by the *ror* instruction. This strategy has not been imple-

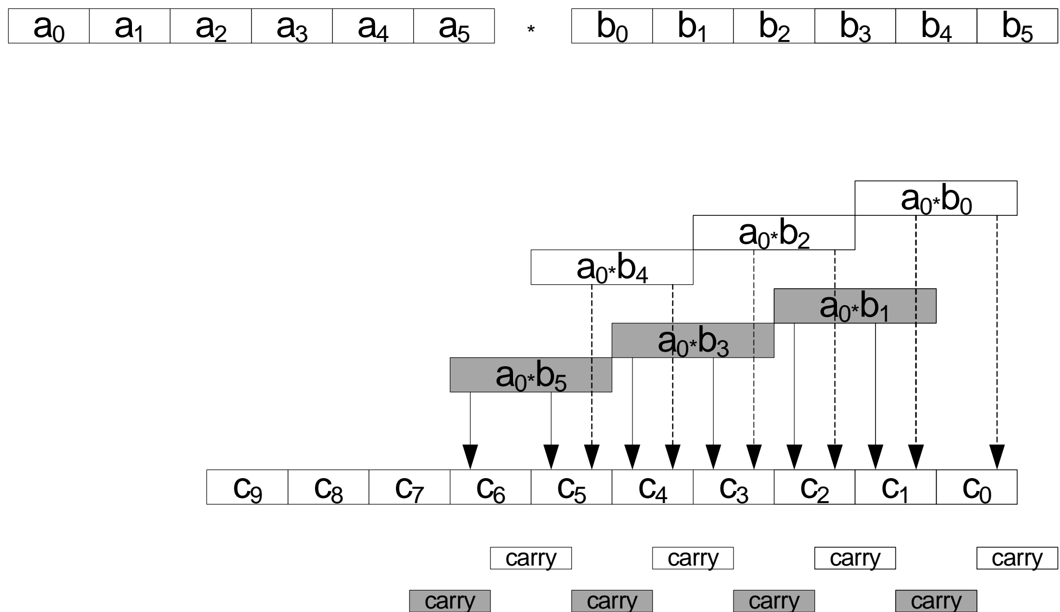

Figure 5.9: Handling Carry Bits by Reordering the Additions

mented, because it requires a more complicated pointer handling. Since no registers are left for pointer storing (due to the large column width) the pointers are sharing registers with the accumulator and that causes stack usage which is equivalent to SRAM usage. This produces a slight overhead which directly scales with a more frequent usage of pointers. Hence, performance gain by the described rearrangement of the blocks could be suboptimal. Instead a completely new approach with a drastically decreased column width was done, as this also promises possibilities for a later re-enrollment of the loops with a moderate overhead, since loop counters need registers to be stored in. A slightly decreased column width might still be a good option, if the code is recommended to be as small as possible. An overhead of two clock cycles per block for handling the carry bits might be an option.

5.2.4 Discussion of the implementation with d=10

The two bottlenecks are estimated by 840 clock cycles for SRAM usage and 1600 clock cycles for carry handling in the worst case. Accumulating this with the minimum effort of 1600 clock cycles results in 4040 clock cycles plus the non estimated overhead for the worst case bottlenecks. The first approach reduced this to 240 clock cycles for SRAM usage and 1200 clock cycles for carry handling with an option to be reduced to 800 clock cycles. Together with the minimal effort this results in 3040 clock cycles or 2640 for the different carry handling. The implementation n results in 3744 clock cycles in total, whereby the non-estimated overhead is about 700 clock cycles. The latter is the sum of various overheads like pointer handling, initialization etc. Especially the pointer handling takes a lot of time, as no pointer can remain in the registers during the hole execution of the 160-bit multiplication in the registers.

The total result is about 50% slower than the benchmarks of SUN Microsystems in [GPW+04]. We think that this difference is caused mainly by handling carry bits. Besides this no sufficient solution could be found for offering the possibility of re-enrolling of the loops. 400 iterations would have to be packed to a loop, this can only be done with the help of a counter kept in two registers. Counting the iterations and checking for the termination condition thus would consum about four clock cycles per iteration, increasing the minimum effort by 1600 cycles or 100% respectively.

For all these reasons a completely new approach with a much smaller column width is done. This way more registers remain unused by the *hybrid multiplication* and offer smarter approaches for optimization of carry bit handling at the cost of SRAM usage. Nevertheless a lot of optimizations were shown and can be reused while shortcomings can be avoided from the beginning.

5.3 Modular Multiplication with Column Width 5

Instead of reducing the SRAM effort to the minimum this approach uses more cycles for SRAM operations. This has several advantages that will overcome the disadvantages of increased SRAM operations. First, this will reduce the overhead caused by pointers to a normal level, as they do not *always* have to be copied from registers to SRAM and back, like it was the the case when using a column width of 10, see

section 5.2.4. The first implementation had to deal with a much bigger overhead concerning this, because no pointer could remain for a long period in the registers. Second this approach is based on several optimizations for carry handling found in the first approach. The most promising strategy seems to arrange the *elementary instruction blocks* in an order that *add with carry* can be used as often as possible. The most efficient way to handle carry bits is not to deal with them! Even though a minimal code size is not them main goal of this thesis, it would be nice to offer a possibility to use loops instead of macros, i.e. the usage of loops should not drastically increase the runtime.

The crucial question is: how to order the EIB, such that there is virtual no carry handling required? The first approach gave the basic idea of arranging the blocks addition optimized for *add with carry*. However it could not be applied to ATmega128L, because of the 8-bit multiply instruction overwriting the carry flag. Two clock cycles would be needed to store and restore the carry flag in the CPU.

5.3.1 Pipelining EIBs to a Full Row

The idea of this optimization is, contrary to store and restore the carry flag between the 8-bit multiplications, that several *elementary instruction blocks* could be executed *pipelined*. This is done by first performing several multiplications and afterwards adding their results in the accumulator using *add with carry*. Figure 5.10 shows the work flow of the idea. This way only at the end of a series of executions a

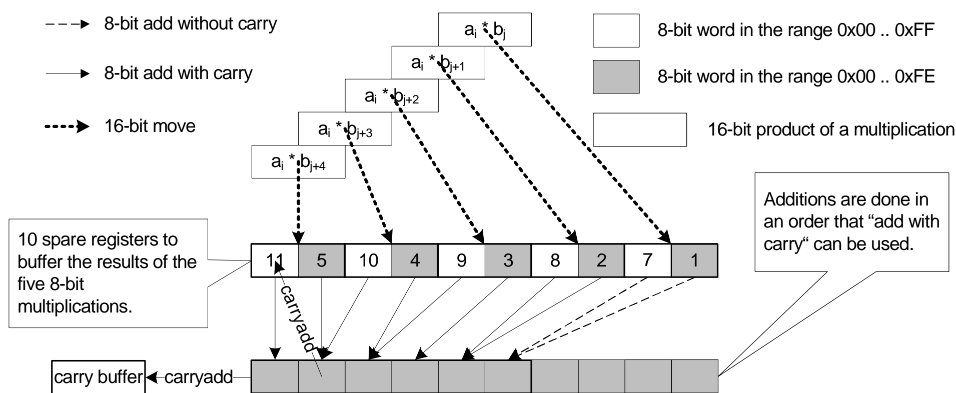

Figure 5.10: Pipelining Elementary Instruction Blocks

carry bit has to be handled. Obviously, this method scales better the more blocks are

pipelined and therefore with more registers occupied. On the other hand this is also true for the column wise multiplication. However only 16 of the 32 registers were reserved for the *hybrid multiplication* which left another 16 registers for pointers pipelining and everything else. Unfortunately the 8-bit multiplications destination registers are not flexible. They are fixed to R0 and R1. Then again the *movw* instruction offers a 2-word move in one clock cycle. Hereby the pipelining can be realized with one additional clock cycle per *elementary instruction block* execution plus the carry handling at the end of the series.

The code for a single row looks as follows:

```
1   mul a0,b
2   movw 14, 0
3   mul a1, b
4   movw 16, 0
5   mul a2, b
6   movw 18, 0
7   mul a3, b
8   movw 26, 0
9   mul a4, b
10  ld b, z+
11
12  add c0, 14
13  adc c1, 16
14  adc c2, 18
15  adc c3, 26
16  adc c4, 0
17
18  adc 1, zero //this is a secure carry add, because it cannot produce another
       carry
19
20  adc c1, 15
21  adc c2, 17
22  adc c3, 19
23  adc c4, 27
24  adc c5, 1
25  /* blabla
26  dflijgög
27  */
28  in carry1, 0x3F    //store SFLAG
29  //teyt
30  ;test
31  andi carry1, 0x01  //mask the carry bit
```

Note that this code does not handle carry bits occurring in the last row of a column, see the following section for detailed information. In this way the overhead for handling the carry bits is six cycles for five EIBs. That means for one EIB the carry handling overhead is reduced to 1,2 cycles. This reduces the overhead of the EIB's effort from 75% in the first implementation to now 30%. Taking one cycle as the theoretical possible minimum overhead this is close to the optimum. The advantage in saving both time and energy is enormous since the EIB is repeated 400 times.

5.3.2 Pipelining Rows to a Full Column

The best method from the first approach of dealing with carry bits is applied to the second approach: extensive use of *add with carry*. As we have emphasized above, this requires the usage of MOVW instructions. To deal with the carry bits that occur at a series another optimizations is used: *secure carry add*. Figure 5.11 shows how this is applied. The figure shows the execution of a full COLUMN WISE

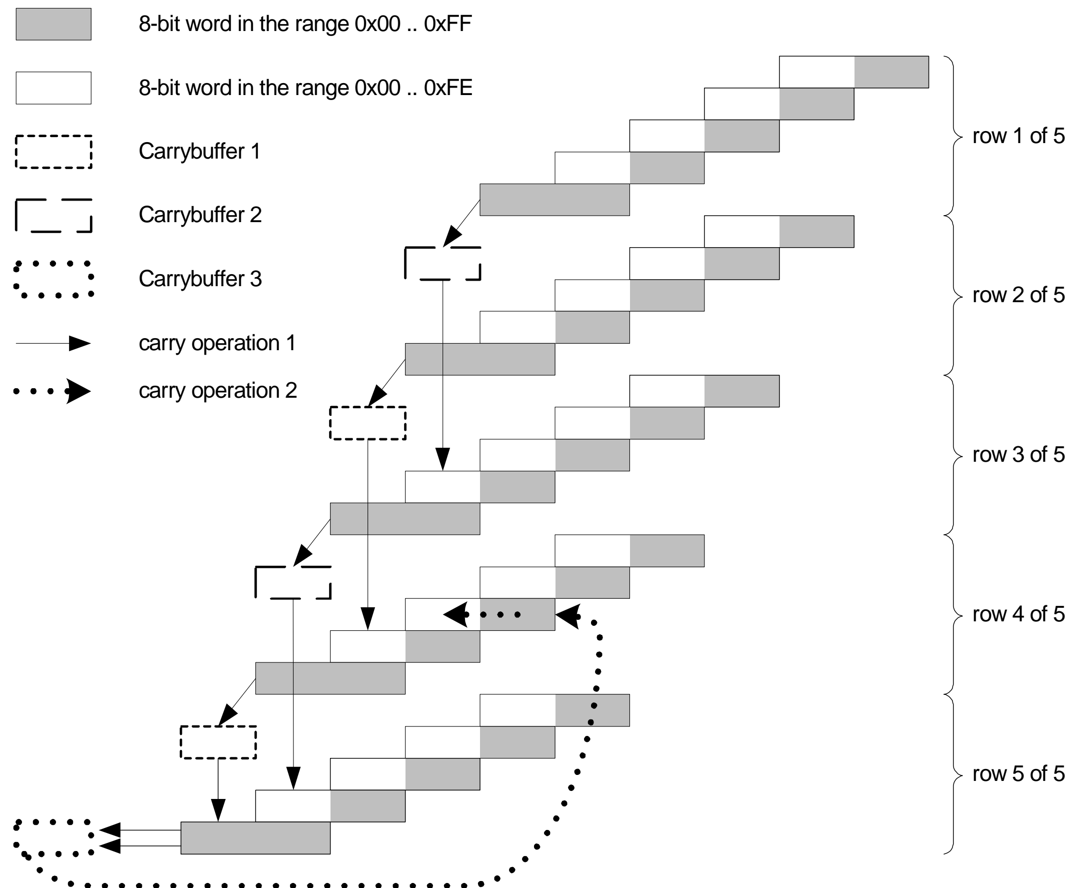

Figure 5.11: Unrolled Column: Pipelining five Rows for Efficient Carry Handling

MULTIPLICATION iteration sequence of the *hybrid multiplication* with a column width of $d = 5$. Since the column width is equivalent to the amount of multiplications executed per column this results in five multiplications in the inner part of the column wise multiplication, which in turn is performed five times. In total this covers 25 *elementary instruction blocks* and is equivalent of performing one column

of the *hybrid multiplication*. 16 columns have to be performed to complete a full 160-bit multiplication ($16 * 5 * 5 = 400$), respectively 400 EIBs.

As the pipelining of blocks to a column already used *secure carry adds*, not every remaining 16-bit product may be available for a secure carry add. White fields show those that are suitable. When the first column is executed the accumulator is still zero and it is possible to skip all buffering of carry bits described in the following. As this is not utilized yet further improvement could be applied here.

However in general the accumulator is not zero and the last addition of a column may produce a carry bit. This is buffered to a register named "Carrybuffer 1" as shown in Figure 5.11. The next possible word to handle this carry is the last added word of the next row, but this word has already been used for a *secure carry add*, see Figure 5.10. Therefore the buffer has to remain untouched until the the execution of the row after the next row, where a *secure carry add* is possible. Since "Carrybuffer 1" is still in usage during the second row, the carry bit occurring at the end is buffered to a register named "Carrybuffer 2". These two buffers hold the occurring carry bit in alternating order.

"Carrybuffer 3" is required at the end of each column. The column carry bits can be accumulated until a column with new product accumulator target starts. A direct add of the "Carry buffer 3" to the accumulator could produce a carry bit and adding this carry bit to the next word, again, might produce a carry bit and so forth. A *secure carry add* is hence required in the next column. Several possibilities in the next column are suitable for this. Figure 5.11 shows the correct position, whereas any white field in the same row would also be possible. The position of this add can be followed at Figure 5.5, which shows the arrangement of the columns and rows of the *hybrid multiplication* for a column width of $d = 5$. The code for a full row macro can be found in appendix A.

5.3.3 Discussion of the Implementation with d=5

This approach divides the available registers of the micro controller for two purposes: reduce SRAM usage and reduced carry bit handling. The breakdown by bottlenecks seems to fit to a column width of $d = 5$ as this way a complete row could be pipelined. With a bigger column width the operands might not fit to the registers and the overhead caused by swapping register to SRAM could cancel out the advantage of the bigger column width though both, the carry handling and the SRAM usage of the *hybrid multiplication*, would profit from a bigger column width. However, this cannot be definitely said, but the analysis of the *schoolbook method* did show, for this

specific case, a theoretical minimum effort of 1600 cycles. The clock cycles consumed by the main bottlenecks carry handling and SRAM usage could be reduced to 400 cycles! In total this result can compete with the, to the best of our knowledge, fastest implementation known so far. Table 5.3 gives an overview of the required instructions.

Mnemonic	**Instr.**	**#C/I**	**Cycles**	**theor. Min.**	**SRAM**	**Carry**
mul	400	2	800	800	0	0
add/adc	986	1	986	800	0	186
in	64	1	64	0	0	64
andi	64	1	64	0	0	64
lds/ld	238	2	476	0	320	0
st	40	2	80	0	80	0
clr	56	1	56	0	0	5
mov/movw	355	1	355	0	0	320
Sum 1			**2881**	**1600**	**400**	**639**
Sum 2			**2881**	**1600**	**1039**	
Sum 3			**2881**	**2639**		

Table 5.3: Required Instructions and Clock Cycles for a 160-bit Multiplication with $d = 5$

Table 5.3 lists the instructions used by theoretical minimum effort, overhead for SRAM usage, and overhead for handling carry bits. Furthermore the in total executed instructions are listed. Column one names the instruction and column two lists the number of these instructions used in total. Column three shows the cycles needed per instruction. In column four the consumed cycles in total are listed. The fifth column recalls the minimum effort while in column six and seven the overhead is described. Column six shows the overhead for SRAM operations to get operands from SRAM and write the product back. The overhead for handling carry bits is shown in the last row. **Sum 1** contrasts the clock cycles required for the total number instructions with the number of instructions used for minimum effort, SRAM usage, and carry handling. In **Sum 2** the total number of instructions is confronted with the minimum effort and total overhead from the two bottlenecks. Finally the minimum effort together with the bottlenecks is confronted with the total number of consumed clock cycles. In **Sum 3** 38.5% of the total overhead (1039) clock cycles are required for SRAM operations, while the other 61.5% are caused by handling carry bits. The ratio between total number of clock cycles and overhead is 0.36.

5.3.4 Loops

The here presented implementation does not include a single loop, but in general could be applied without raising the needed clock cycles significantly. Appendix A, shows the complete code. The main function is divided by code comments into 16 series of macro-calls, each calculating one column. Each of the series calls three macros and in addition seven accumulator manipulating macro-calls appear in total. This is $16 \cdot 3 + 7 = 55$. Realized by jumps this would result in 110 clock cycles plus a small overhead for handing over parameters. In relation to the total amount of clock cycles this is almost negligible, but would significantly reduce the code size of this part of the implementation. Note that the smallest macro only executes three instructions. As in this case the overhead caused by the loop is immense and the code size reduction is low it may be better not to realize this macro with a loop. So even with loops instead of macros the implementation of modular multiplication would remain slightly faster than the one presented in [GPW$^+$04].

5.4 Reduction

According to the explained method in Section 4.3.5 the reduction can be done by additions. Unfortunately adding eight bits starting at bit 31 is not equivalent to an 8-bit add instruction, because bit 31 is bit seven of word four respectively bit seven of a_4. The next seven bits are the first seven bits of a_5. Each word has to be split in two words and the bits have to be shifted to the correct position such that the 8-bit ADD instruction is adding the bits to the correct position. However besides this a second bottleneck appears: carry bits. Performing any of the resulting 8-bit ADD instruction requires to add a carry bit to the most significant byte, where again a carry bit may be produced and so on. Reaching bit 160 reduces the carry bit and it is added then to bit 0 and 31. To be added to bit 31 the buffer in which the carry bit is stored has to be shifted. Each of these two additions again starts a carry add chain. The chain finally terminates at the word where the first addition was done. Of course this worst case scenario scales pretty badly, but usually just a few additions are necessary. Therefore checking if a carry bit actually occurs is useful when performing reductions, contrary to multiplication.

Table 5.4 displays the formula for each byte of a 160-bit multiplication that has to be reduced. a_{36} is a special case. It is reduced like $a_{36.0} * 2^{288} = a_{36} * 2^{159} + a_{36} * 2^{128}$ but this is only true for bit zero of a_{36}. Summand one of this reduction formula hast be reduced again for bit one to seven resulting in $a_{36.1..7} * 2^{288} = a_{36} * 2^{31} + a_{36} * 2^{0} + a_{36} * 2^{128}$.

a_{20}	$*$	2^{160}	$=$	a_{20}	$*$	2^{31}	$+$	a_{20}	$*$	2^{0}				
a_{21}	$*$	2^{168}	$=$	a_{21}	$*$	2^{39}	$+$	a_{21}	$*$	2^{8}				
a_{22}	$*$	2^{176}	$=$	a_{22}	$*$	2^{47}	$+$	a_{22}	$*$	2^{16}				
a_{23}	$*$	2^{184}	$=$	a_{23}	$*$	2^{55}	$+$	a_{23}	$*$	2^{24}				
a_{24}	$*$	2^{192}	$=$	a_{24}	$*$	2^{63}	$+$	a_{24}	$*$	2^{32}				
a_{25}	$*$	2^{200}	$=$	a_{25}	$*$	2^{71}	$+$	a_{25}	$*$	2^{40}				
a_{26}	$*$	2^{208}	$=$	a_{26}	$*$	2^{79}	$+$	a_{26}	$*$	2^{48}				
a_{27}	$*$	2^{216}	$=$	a_{27}	$*$	2^{87}	$+$	a_{27}	$*$	2^{56}				
a_{28}	$*$	2^{224}	$=$	a_{28}	$*$	2^{95}	$+$	a_{28}	$*$	2^{64}				
a_{29}	$*$	2^{232}	$=$	a_{29}	$*$	2^{103}	$+$	a_{29}	$*$	2^{72}				
a_{30}	$*$	2^{240}	$=$	a_{30}	$*$	2^{111}	$+$	a_{30}	$*$	2^{80}				
a_{31}	$*$	2^{248}	$=$	a_{31}	$*$	2^{119}	$+$	a_{31}	$*$	2^{88}				
a_{32}	$*$	2^{256}	$=$	a_{32}	$*$	2^{127}	$+$	a_{32}	$*$	2^{96}				
a_{33}	$*$	2^{264}	$=$	a_{33}	$*$	2^{135}	$+$	a_{33}	$*$	2^{104}				
a_{34}	$*$	2^{272}	$=$	a_{34}	$*$	2^{143}	$+$	a_{34}	$*$	2^{112}				
a_{35}	$*$	2^{280}	$=$	a_{35}	$*$	2^{151}	$+$	a_{35}	$*$	2^{120}				
$a_{36.0}$	$*$	2^{288}	$=$	a_{36}	$*$	2^{159}	$+$	a_{36}	$*$	2^{128}				
$a_{36.1..7}$	$*$	2^{288}	$=$	a_{36}	$*$	2^{31}	$+$	a_{36}	$*$	2^{0}	$+$	a_{36}	$*$	2^{128}
a_{37}	$*$	2^{296}	$=$	a_{37}	$*$	2^{38}	$+$	a_{37}	$*$	2^{7}	$+$	a_{37}	$*$	2^{136}
a_{38}	$*$	2^{304}	$=$	a_{38}	$*$	2^{46}	$+$	a_{38}	$*$	2^{15}	$+$	a_{38}	$*$	2^{144}
a_{39}	$*$	2^{312}	$=$	a_{39}	$*$	2^{54}	$+$	a_{39}	$*$	2^{23}	$+$	a_{39}	$*$	2^{152}

Table 5.4: Reduction of a 320-bit Integer

Recall that the reduction can be done by shifts and additions. Assuming a 320-bit integer that has to be reduced, than a map can be created that shows for each of the 160-most significant bits the correct bits in the least significant 160-bits to be added to. Figure 5.12 illustrates the exact bit positions. The top row displays the 20 most significant bytes, of the 320-bit word that has to be reduced. The numbers in the range from 0 to 7 that can be found in the rows address the bit of the top rows byte. The left most column then shows the according destination bit. Each bit of the most significant 20 bytes may be added to multiple bit of the least significant 20 bytes. Of course 8-bit are to be added at once if possible.

	mso	a20	a21	a22	a23	a24	a25	a26	a27	a28	a29	…	a36	a37	a38	a39				…	a26	a27	a28	a29	a30	a31	a32	a33	a34	a35	a36	a37	a38	a39
lso																																		
a09	79							0			7							a19	159												0			
a09	78						7				6							a19	158											7				
a09	77						6				5							a19	157											6				
a09	76						5				4							a19	156											5				
a09	75						4				3							a19	155											4				
a09	74						3				2							a19	154											3				
a09	73						2				1							a19	153											2				
a09	72						1				0							a19	152											1				
a08	71						0			7								a18	151											0			7	
a08	70					7				6								a18	150										7				6	
a08	69					6				5								a18	149										6				5	
a08	68					5				4								a18	148										5				4	
a08	67					4				3								a18	147										4				3	
a08	66					3				2								a18	146										3				2	
a08	65					2				1								a18	145										2				1	
a08	64					1				0								a18	144										1				0	
a07	63					0			7									a17	143										0			7		
a07	62				7				6									a17	142									7				6		
a07	61				6				5							7		a17	141									6				5		
a07	60				5				4							6		a17	140									5				4		
a07	59				4				3							5		a17	139									4				3		
a07	58				3				2							4		a17	138									3				2		
a07	57				2				1							3		a17	137									2				1		
a07	56				1				0							2		a17	136									1				0		
a06	55				0			7								1		a16	135									0			7			
a06	54			7				6								0		a16	134								7				6			
a06	53			6				5							7			a16	133								6				5			
a06	52			5				4							6			a16	132								5				4			
a06	51			4				3							5			a16	131								4				3			
a06	50			3				2							4			a16	130								3				2			
a06	49			2				1							3			a16	129								2				1			
a06	48			1				0							2			a16	128								1				0			
a05	47			0			7								1			a15	127								0			7				
a05	46		7				6								0			a15	126							7				6				
a05	45		6				5							7				a15	125							6				5				
a05	44		5				4							6				a15	124							5				4				
a05	43		4				3							5				a15	123							4				3				
a05	42		3				2							4				a15	122							3				2				
a05	41		2				1							3				a15	121							2				1				
a05	40		1				0							2				a15	120							1				0				
a04	39		0			7								1				a14	119							0			7					
a04	38	7				6								0				a14	118						7				6					
a04	37	6				5							7					a14	117						6				5					
a04	36	5				4							6					a14	116						5				4					
a04	35	4				3							5					a14	115						4				3					
a04	34	3				2							4					a14	114						3				2					
a04	33	2				1							3					a14	113						2				1					
a04	32	1				0							2					a14	112						1				0					
a03	31	0			7								1					a13	111						0			7						
a03	30				6											7		a13	110					7				6						
a03	29				5											6		a13	109					6				5						
a03	28				4											5		a13	108					5				4						
a03	27				3											4		a13	107					4				3						
a03	26				2											3		a13	106					3				2						
a03	25				1											2		a13	105					2				1						
a03	24				0											1		a13	104					1				0						
a02	23			7												0		a12	103					0			7							
a02	22			6											7			a12	102				7				6							
a02	21			5											6			a12	101				6				5							
a02	20			4											5			a12	100				5				4							
a02	19			3											4			a12	99				4				3							
a02	18			2											3			a12	98				3				2							
a02	17			1											2			a12	97				2				1							
a02	16			0											1			a12	96				1				0							
a01	15		7												0			a11	95				0			7								
a01	14		6											7				a11	94			7				6								
a01	13		5											6				a11	93			6				5								
a01	12		4											5				a11	92			5				4								
a01	11		3											4				a11	91			4				3								
a01	10		2											3				a11	90			3				2								
a01	9		1											2				a11	89			2				1								
a01	8		0											1				a11	88			1				0								
a00	7	7												0				a10	87			0			7									
a00	6	6											7					a10	86		7				6									
a00	5	5											6					a10	85		6				5									
a00	4	4											5					a10	84		5				4									
a00	3	3											4					a10	83		4				3									
a00	2	2											3					a10	82		3				2									
a00	1	1											2					a10	81		2				1									
a00	0	0											1					a10	80		1				0									

Figure 5.12: Reduction of a 160-bit Integer

5.5 Bisection

The bisection is realized according to Algorithm 8. The shifts are done using the rotate instruction. This way the bit rotated out of the first byte is rotated into the next byte with the next shift. The rotations start with the most significant byte. The last rotate puts the least significant bit into the carry flag. As this bit is an indicator for the integer being odd or even a branch instruction can initialize a branch to a point behind the additions in case the integer is even. In case the integer is odd, $\lceil p >> 1 \rceil$ has to be added to the results. Since SRAM operation are slow $\lceil p >> 1 \rceil$ is loaded byte by byte to a temporary register and added to the result using the add with carry instruction. The assembly code is listed below.

```
1   ror 22
2   ror 23
3   ror 24
4   ror 25
5   ror 26
6   ror 27
7
8   BRCC evenitis
9   //if carry is set, the number to be halfed was odd
10
11  ldi 28, 0x00
12  add 27, 28
13  ldi 28, 0x00
14  adc 26, 28
15  ldi 28, 0x00
16  adc 25, 28
17  ldi 28, 0xC0
18  adc 24, 28
19  ldi 28, 0xFF
20  adc 23, 28
```

5.6 Addition and Subtraction

These longterm arithmetics deal easily with issues like carry bits, smart operand fetching to reduce SRAM usage, and reduction to become a prime field arithmetic.

Carry bits are easy to handle, as *add with carry* respectively *subtract with carry* handle the problem. If after the addition/subtraction the carry flag is set, simply subtract/add the modulo p. As every word of the operand is used only once, no special operand fetching mechanism is required.

5.7 Inversion

Contrary to all other components of the prime field arithmetic, this module was not implemented in assembly, but rather in C. This is because inversion is only needed once per point multiplication on the elliptic curve. The only reason to implement it in assembly would be to reduce code size.

6 Results

This chapter summarizes the results concerning speed and energy consumption.

6.1 Speed

The basic requirement for a fast and thus energy efficient implementation of ECC is a very fast multiplication in the prime field. The fastest known implementation was implemented by SUN Microsystems. In [GPW+04] they provide a benchmark for the micro controller that we used as well, hence a direct comparison is possible. A 160-bit multiplication from SUN Microsystems' implementation requires 3106 cycles, which is at a clock rate of 7.37 MHz equivalent to 0.42 ms.

The implementation presented in this work needs 2881 cycles for a 160-bit multiplication, which is equivalent to 0.39 ms at 7.37 MHz. In fact, this represents a time saving of 7.2%. To the best of our knowledge this is the fastest implementation world wide of a modular multiplication of a 160-bit standardized elliptic curve for an 8-bit micro controller.

In Table 6.1 we present a detailed list of instructions used by our and SUN Microsystems' implementation as published in [GPW+04]. In the bottom line there are three bold marked total clock cycle numbers: the left bold marked sum (3744) represents the total amount of required clock cycles for a 160-bit multiplication using the hybrid multiplication with a column width of $d = 10$. The aim of this approaches was to highly reduce the SRAM access, as can be seen by the small amount of ld an st instructions, respectively. Compared to the other two approaches a significant saving in this point is achieved. Since the hybrid multiplication with a colum width of $d = 10$ requires nearly all available registers no sufficient carry handling method could be applied here, what causes the slow overall performance of this approach. In addition a minimum amount of clock cycles is given for this column width that contains the minimum effort plus the SRAM usage of the hybrid multiplication with

a column width of $d = 10$. The overhead of the total amount of required clock cycles and the minimal amount of clock cycles is 103%.

The second bold sum (2881) represents the second approach of this work, with a column width of $d = 5$. Compared with the first approach a bigger amount of SRAM operations is used. Also compared to the implementation of SUN Microsystems (3106) more SRAM operations are required. The difference to the first approach is due to the bigger column width, this can be seen when comparing the minimum amount of clock cycles required for the column width of $d = 5$ with that from $d = 10$. The overhead of the total amount of required clock cycles and the minimal amount of clock cycles is 44%. The overhead of the SUN Microsystems' implementation and the minimal amount of clock cycles is 55%. As one can see in Table 6.1, the main differences between our implementation and SUN Microsystems' lie in the number of used additions and data loads. Note that data loads require two cycles contrary to the addition, which only requires one cycle. Although SUN Microsystems' implementation executes less data load instructions, in total it requires more cycles than our implementation. The time saving results from the improved carry handling reducing the number of needed additions close to the minimum of 800 for a column width of 5. In SUN Microsystems' implementation the number of data loads is close to the minimum number of 160 data loads. The additional data loads in our implementation result from pointer handling. Pointers have to be restored from SRAM very often, because the carry handling needs all spare registers.

instr	**#C/I**	**d=10**		**d=5**		
		min	**this work**	**min**	**this work**	**SUN**
mul	2	800	800	800	800	800
add	1	800	1360	800	986	1360
in	1		400		64	
andi	1		400		64	
ld	2	160	176	320	476	334
st	2	80	80	80	80	80
clr	1				56	
movw	1				355	335
other			528			197
SUM		1840	**3744**	2000	**2881**	**3106**

Table 6.1: Comparison of Executed Instruction for one 160-bit Multiplication

The code size of the 160-bit multiplication is 5.24 KB. As it is highly optimized for speed, this is bearable. Furthermore 112 B of SRAM are required to store operands and pointers.

Comparison with TinyECC is cumbersome for two reasons: on the one hand neither time tables for curve nor modular arithmetic for TinyECC are available. On the other hand we did not implement a full ECDSA protocol. Therefore we estimate the execution time of an ECDSA signature based on our modular multiplication. The elliptic curve implemented in C requires 1.06s to perform a point multiplication. It uses the highly optimized modular arithmetic but to the implementation of the curve more optimizations can be applied. [GPW+04] state that 77% of the execution time of one point multiplication is required for modular multiplication. Assuming our multiplication to be used here would result in 0.76s. Note that this curve arithmetic includes some well applied algorithmic optimizations which are best fitted to hardware, because they are done in assembly. On the other hand no special optimization for ECDSA were included, e.g. the y-coordinate is calculated but not required for the ECDSA protocol. A signature requires one inversion, two modular multiplication, and one modular addition. In addition one SHA-1 has to be executed to hash the message. Generally SHA-1 and a modular multiplication are both roughly three orders of magnitude faster than a point multiplication. The execution time of an inversion is in the range of several modular multiplications. The execution time of the modular addition is roughly four orders of magnitude faster than the execution of a point multiplication. Therefore, we estimate that all required operations for an ECDSA signature, including the SHA-1, can most probably be performed in less than one second. A TinyECC ECDSA signature generation takes slightly less than two seconds, including the time for the SHA-1 execution. Furthermore, once a precomputation time of 3.5s is required.

6.2 Energy

We assume that the MICAz respectively its micro controller ATmega128L consumes 4.88 nWs per clock cycle. According to [PLP06], the MICAz can use 31.25% of the batteries total capacity respectively 6750 Ws. This is due to the fact that the voltage of a battery pack drops while discharging and when voltage drops below a certain point the ramaing energy cannot be used. Since a 160-bit multiplication requires 2881 clock cycles the hereby consumed energy is 14 μWs. That would be 480 million multiplications in a nodes lifetime.

Table 6.2 shows the estimated energy consumptions. From left-to-right are displayed: the here presented 160-bit multiplication with a column width of $d = 10$ followed by the approach with a column width of $d = 5$. Subsequently the elliptic curve multiplications are displayed, starting with the here implemented version followed by that from SUN Microsystems. Finally the approximated curve multiplication is

presented, in which we assumed our 160-bit multiplication being used in the curve of SUN Microsystems.

	160-bit mul			**k*P**		
	d=10	**d=5**	**SUN**	**ECC**	**ECC SUN**	**ECC opt**
time in [ms]	0.51	0.39	0.42	1151	810	760
energy in [nWs]	0.10	0.08	0.09	236	166	156
energy/bit in [pWs]	0.65	0.50	0.54	1475	1038	975

Table 6.2: Comparison of Energy Consumption

7 Summary and Future Work

In this chapter we will first summarize the work and subsequently provide a pointer to future work.

7.1 Summary

In Chapter 1 we explained that energy consumption and security are crucial for the integration of *ubiquitous* and *pervasive* computing. We describe that asymmetric algorithms come at higher computational cost, but have significant advantages in key establishment and distribution. Subsequently, in Chapter 2 we gave an introduction to *WSNs*, the *MICAz* node, and the *MICAz's* micro controller ATmega128L. The processor is able to perform multiplications in two clock cycles, which is besides additions and memory accesses the most important instruction when implementing long term multiplication. We assumed an energy consumption of 4.88 nWs per clock cycle for the micro controller clocked at 7.37 MHz. Furthermore we describe the different tools used for the implementation. A combination of *WinAVR* and *AVRStudio* allows to implement projects in C and GNU assembly.

We then gave a short introduction to cryptography and the mathematics required in Chapter 3. Furthermore we compared existing implementations of asymmetric algorithms to show that elliptic curves are the favorites, due to their high speed and short key length. Besides this, elliptic curve cryptography is considered to be safe, because it is known since 1986 and protocols like ECDSA, which base on the discrete logarithm problem, have not been broken until now. Subsequently, we divided ECC systems into the three parts prime field arithmetic, curve arithmetic, and protocol layer. Optimizations and selection of algorithms on every three levels are relevant for the speed. An efficiently implemented prime field can be reused for several implementations of the curve, which is not true for the curves' optimization that has at least to be partly recoded for usage with other protocols. We therefore focused on an efficient modular arithmetic, because the 160-bit multiplication is the most time consuming part.

In the following main part of the work we presented a prime field for a standardized curve implemented in assembly, which has to serve as basis for future highly efficient implementations of the elliptic curve for different protocols. A careful choice of algorithms and optimizations, as well as smart handling of carry bits resulted, to the best of our knowledge, in the fastest known implementation of the 160-bit multiplication of the prime field secp160r1 for the micro controller ATmega128L. The implementation is open source and available on request. An elliptic curve was implemented in C, which contains some substantial optimizations that may be reused for highly efficient implementation.

Finally, in Chapter 6 we compared the implemented prime body and the implemented curve. The prime field is approximately 7% faster than the fastest known so far. It requires 2881 clock cycles, respectively 0.39 ms for a 160-bit multiplication. Even though the implementation is highly optimized for speed, the code-size of 5.4 KB and RAM requirements of 112 B are acceptable. The elliptic curve provides only basic optimizations and performs a point multiplication in 1.151 s. The prime field can be integrated in different implementations of the concerning elliptic curve without the need of applying new optimizations. A highly optimized curve using the here provided primefield could execute the curve multiplication in 760 ms and would therefore allow an ECDSA signature in less than one second.

7.2 Future Work

The next steps could be the implementation of a full optimized elliptic curve for a specific protocol based on EC like ECDSA. Furthermore this could then be integrated in TinyOS and therefore be ported to NesC. For use in TinyOS it might be considerable to apply loops as described in Section 5.3.4. Nevertheless, as the project is supposed to become open source, a project homepage should be developed and hosted.

Besides this it would be most interesting to apply further optimizations to the prime field. It might be worth a try to encapsulate the long term multiplication in a Karatzuba command and conquer. At first glance it might not occur useful, as Karatzuba-Ofman multiplication switches one multiplication to four additions. E.g. considering x and y as 160-bit integers, then $x * y$ could be performed by three 80-bit multiplications and two additions and two subtractions. Of course the 80-bit multiplications could again be multiplied this way. However for one recursion this would change 400 8-bit multiplications to 300 8-bit multiplications, two 80-bit additions, and two 80-bit subtractions, which is equivalent to 320 add/sub instructions. This

results in trading 100 multiplications to 320 add/sub instructions, which results in a slow down - in theory!

On the other hand the multiplications has to be accumulated and therefore one could also speak of switching 100 multiplications and 200 additions (which will probably produce a remarkable overhead by carry handling) to 320 additions/subtractions (which might not produce that large overhead by handling carry bits). A closer look at this seems to be most interesting.

How to split operand with Karatsuba-Ofman and which strategy to get operands from SRAM and accumulator back to SRAM represents the second bottleneck. Different approaches might be necessary to solve both bottlenecks sufficient and release efficient applications. In summary it is not really about saving multiplications, as the AVR micro controller can perform a multiplication in two clock cycles, but rather about an option to get rid of the register consuming and carry bit producing accumulations following the multiplications.

However it would be nice to describe the here applied strategy of handling carry bits in a general long-term-integer-multiplication algorithm. Further research and more implementations with different bit length and different hardware would be needed for this.

Finally the implementation of the curve should be further improved. The arithmetic in mixed coordinates can be reused unless one considers to implement the whole curve arithmetic in assembly. The variety of additional improvements is large and partly depending on the used protocol. Before using the elliptic curve, that was implemented in this thesis, the modular reduction should be called from the assembly program that implements the multiplication, rather than doing this call in the C program. This way the variables used for the arithmetic on the elliptic curve could be halved in size, as they are not forced anymore to temporarily save the 320-bit result from the 160-bit multiplication.

Bibliography

[Bro01] Brown, M and Hankerson, D. and López, J. and Menezes, J. Software Implementation of the NIST Elliptic Curves Over Prime Fields. *Lecture Notes in Computer Science*, 2020:250ff, 2001.

[CMO98] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. *Advances in Cryptology–ASIACRYPT*, 98:51–65, 1998.

[Cora] Atmel Corporation. 8-bit Microcontroller with 128K Bytes In-System Programmable Flash, Revision 2467N-AVR-03/06.

[Corb] Atmel Corporation. AVR Product Line Introduction, Presentation by Atmel. `http://www.atmel.com/products/avr/overview.asp`.

[Corc] Atmel Corporation. AVR Studio, 4.12.490, Service Pack 3. `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725`.

[Cord] Atmel Corporation. Development Tools User Guide - Section 4 - AVR Assembler User Guide. `http://www.atmel.com/dyn/products/other_docs.asp?family_id=607#User\%20Guide`.

[CPS03] H. Chan, A. Perrig, and D. Song. Random Key Predistribution Schemes for Sensor Networks. In *Proceedings of the IEEE Security and Privacy Symposium 2003*, 2003.

[DDHV] W. Du, J. Deng, Y. Han, and P. Varshney. A Pairwise Key Predistribution Scheme for Wireless Sensor Networks. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*.

[EG02] L. Eschenauer and V. Gligor. A Key Management Scheme for Distributed Sensor Networks. In *CCS '02: Proceedings of the 9th ACM*

Conference on Computer and Communications Security, New York, NY, USA, 2002. ACM Press.

[GB] GNU-Binutils. collection of binary tools: GNU linker (ld) and assembly (as). http://www.gnu.org/software/binutils/.

[GLvB+03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

[GPW+04] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004), 6th International Workshop, pages 119–132*, 2004.

[HC02] J.L. Hill and D.E. Culler. Mica: a Wireless Platform for Deeply Embedded Networks. *Micro, IEEE*, 22(6):12–24, 2002.

[HMV04] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, New York [et al.], 2004.

[HSW+00] J. Hill, J Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000.

[Inca] Crossbow Technology Inc. MICAz WIRELESS MEASUREMENT SYSTEM, Document Part Number: 6020-0060-03 Rev A. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[Incb] Crossbow Technology Inc. MPR-MIB Users Manual, Document Part Number: 7430-0021-07 Revision B. http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf.

[Incc] Crossbow Technology Inc. Wireless Sensor Networks. http://www.xbow.com/.

[LN06] An Liu and Peng Ning. TinyECC: Elliptic Curve Cryptography for Sensor Networks. available for download at http://discovery.csc.ncsu.edu/software/TinyECC, September 2006.

[LV99] A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Key Sizes. November 1999.

[MM04] C. Siva Ram Murthy and B. S. Manoj. *Ad hoc wireless networks.* Prentice Hall, Upper Saddle River, NJ, 2004.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, Florida, USA, first edition, 1997.

[MWS04] D. Malan, M. Welsh, and M. Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography, 2004.

[oES] Harvard University School of Engineering and Applied Sciences. Codeblue- wireless sensor networks for medical care.

[oss] open source software. Programmers's Notepad - text editor with special features for coders. `http://www.pnotepad.org/`.

[PLP06] K. Piotrowski, P. Langendoerfer, and S. Peter. How public key cryptography influences wireless sensor node lifetime. In *SASN '06: Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, pages 169–176, New York, NY, USA, 2006. ACM Press.

[proa] Free Software project. AVR Libc - high quality C library for use with GCC on Atmel AVR microcontrollers. `http://www.nongnu.org/avr-libc/`.

[Prob] GNU Project. GNU Compiler Collection. `http://gcc.gnu.org/`.

[Res00] Certicom Research. SEC 1: Elliptic Curve Cryptography, Version 1.0, September 2000.

[Sch96] B. Schneier. *Applied cryptography.* Wiley, New York [et al.], second edition, 1996.

[WAR06] Y. Wang, G. Attebury, and B. Ramamurthy. A survey of security issues in wireless sensor networks. *Communications Surveys & Tutorials, IEEE*, 8(2):2–23, 2006.

[WGE+05] A.S. Wander, N. Gura, H. Eberle, V. Gupta, and S.C. Shantz. En-

ergy Analysis of Public-key Cryptography for Wireless Sensor Networks. *Third IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, 2005.

[Win] WinAVR. Suite of executable, open source software development tools for the Atmel AVR series of RISC microprocessors hosted on the Windows platform. `http://winavr.sourceforge.net/`.

[WKC+04] R. Watro, D. Kong, S. F. Cuti, C. Gardiner, C. Lynn, and P. Kruus. TinyPK: Securing Sensor Networks with Public Key Technology. In *SASN '04: Proceedings of the 2nd ACM Workshop on Security of Ad Hoc and Sensor Networks*, pages 59–64, New York, NY, USA, 2004. ACM Press.

A Source Code

```
.MACRO rowwise_multiplication newcolumn
  eor zero, zero // refresh zero with "0"

  /// b0
  mul a0,b
  movw 14, 0
  mul a1, b
  movw 16, 0
  mul a2, b
  movw 18, 0
  mul a3, b
  movw 26, 0
  mul a4, b
  ld b, z+

  add c0, 14
  adc c1, 16
  adc c2, 18
  adc c3, 26
  adc c4, 0

  adc 1, zero //this is a secure carry add, because it cannot provoke another
      carry

  adc c1, 15
  adc c2, 17
  adc c3, 19
  adc c4, 27
  adc c5, 1

  in carry1, 0x3F    //store SFLAG
  andi carry1, 0x01  //mask the carry bit

  /// b1
  mul a0,b
  movw 14, 0
  mul a1, b
  movw 16, 0
  mul a2, b
  movw 18, 0
  mul a3, b
  movw 26, 0
  mul a4, b
  ld b, z+

  add c1, 14
  adc c2, 16
  adc c3, 18
```

```
48     adc c4, 26
49     adc c5, 0
50     adc 1, zero //this is a secure carry add, because it cannot provoke another
          carry
51
52     adc c2, 15
53     adc c3, 17
54     adc c4, 19
55     adc c5, 27
56     adc c6, 1
57     in carry2, 0x3F    //store SFLAG
58     andi carry2, 0x01  //mask the carry bit
59
60
61     /// b2
62     mul a0,b
63     movw 14, 0
64     mul a1, b
65     movw 16, 0
66     mul a2, b
67     movw 18, 0
68     mul a3, b
69     movw 26, 0
70     mul a4, b
71     ld b, z+
72
73     add c2, 14
74     adc c3, 16
75     adc c4, 18
76     adc c5, 26
77     adc c6, 0
78     adc 1, zero //this is a secure carry add, because it cannot provoke another
          carry
79
80     add 27, carry1 //delayed secure carry add
81     add c3, 15
82     adc c4, 17
83     adc c5, 19
84     adc c6, 27
85     adc c7, 1
86     in carry1, 0x3F    //store SFLAG
87     andi carry1, 0x01  //mask the carry bit
88
89     /// b3
90     mul a0,b
91     movw 14, 0
92     mul a1, b
93     movw 16, 0
94     mul a2, b
95     movw 18, 0
96     mul a3, b
97     movw 26, 0
98     mul a4, b
99     ld b, z+
100
101    .if \newcolumn
102    add 18, carry3  // unsecure carry add
103    adc 19, zero    // secure carry add
104    eor carry3, carry3
105    .endif
```

```
106
107     add c3, 14
108     adc c4, 16
109     adc c5, 18
110     adc c6, 26
111     adc c7, 0
112     adc 1, zero //this is a secure carry add, because it cannot provoke another
            carry
113
114     add 27, carry2
115     adc c4, 15
116     adc c5, 17
117     adc c6, 19
118     adc c7, 27
119     adc c8, 1
120     in carry2, 0x3F     //store SFLAG
121     andi carry2, 0x01  //mask the carry bit
122
123     /// b4
124     mul a0,b
125     movw 14, 0
126     mul a1, b
127     movw 16, 0
128     mul a2, b
129     movw 18, 0
130     mul a3, b
131     movw 26, 0
132     mul a4, b
133     ld b, z+
134
135     add c4, 14
136     adc c5, 16
137     adc c6, 18
138     adc c7, 26
139     adc c8, 0
140     adc 1, zero //this is a secure carry add, because it cannot provoke another
            carry
141
142     add 27, carry1
143     add c5, 15
144     adc c6, 17
145     adc c7, 19
146     adc c8, 27
147     adc c9, 1
148     adc carry3, zero //carry3 is the sum of all carrys of one column (up to 3
            rows)
149
150     add c9, carry2 //non-secure carry add
151     adc carry3, zero
152
153 .ENDM
```