Implementing cryptographic solutions in a modular smartcard core
RSA hardware modeling and simulation : elliptic curves design and implementation

Cuppens, W.H.P.; Willemse, G.A.J.

Award date:
2002

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
Implementing Cryptographic Solutions in a Modular SmartCard core

RSA hardware modeling and simulation – Elliptic Curves design and implementation

ing. W.H.P. Cuppens
G.A.J. Willemse

Coaches: ing. H.P.F. Janssen (SafeNet, Vught)
dr.ir. C.J.A. Jansen (SafeNet, Vught)
G. Szanto ba. (SafeNet, Vught)
dr.ir. A.C. Verschueren
Supervisor: prof.ir. M.P.J. Stevens
Examinator: dr.ir. T.J. Tjalkens
Period: September 2001 – June 2002

The Faculty of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of Master’s Theses
Implementing Cryptographic Solutions in a Modular SmartCard core

RSA hardware modeling and simulation - Elliptic Curves design and implementation

CONFIDENTIAL
Implementing Cryptographic Solutions in a Modular SmartCard core

RSA hardware modeling and simulation - Elliptic Curves design and implementation

- RSA
- Elliptic curves
- Montgomery
- Software design
- Hardware design

ing. W.H.P. Cuppens
G.A.J. Willemse

Graduation project,
Eindhoven, University of Technology
September 2001 – June 2002

Supervisors:
SafeNet, Vught
ing. H.P.F. Janssen
dr. ir. C.J.A. Jansen
G. Szanto ba.

Eindhoven, University of Technology
prof. ir. M.P.J. Stevens
dr. ir. A.C. Verschueren
Summary

Background of the project
The use of cryptography becomes more and more important. Messages that need to be send via a public network has to be secure to prevent being read by other parties. Today's most used cryptographic algorithm is RSA. A possible future algorithm is elliptic curve cryptography. The common key-length to have secure communication is 1024 bits using RSA according to 160 bits with elliptic curve cryptography systems. The keys are used to encrypt or decrypt the messages into secure data. Except security, area usage, performance, power dissipation and scalability are important issues designing the cryptography systems.

Problem definition
The graduation project at SafeNet© will study, specify, define and implement two low-cost cryptographic IP cores. The project is divided into two parts, the first part introduced the complexity of cryptography and low-cost designs. It concerned the development of a software model of a new hardware design that performs RSA calculations with Montgomery. The second part involved the designing of a low-cost cryptographic solution using elliptic curve cryptography (ECC). Different ECC solution have to be compared. Choose the best configuration to develop and implement a low-cost hardware design.

Methods of inquiry for the software model
The software model is developed to do cycle accurate simulations of a new low-cost smartcard coprocessor design. The software model is divided in several hierarchical classes similar to the hardware design. A separate relation class model is developed that indicates the signals and communication channels. The software model has the opportunity to trace all registers and signals of processing elements, register-bank and control-module down to bit-level. For the development of the software model Borland C++ Builder 5.0™ is used.

Results and conclusions of the development of the software model
The model can be used for internal comprehension of the product as well as for product information towards customers. With the model exact performance estimations for different configurations are realized in an early development stage. Furthermore the software model gives the opportunity to simulate calculations and validate the hardware implementation. It generates inputs and outputs that are useful to run and check the hardware simulations. The simulations with the model are performed factors faster than in hardware, a lot of border-case situations can be verified on beforehand. VHDL command files are generated for every hierarchical level. The software model turned out to be very useful for internal and external comprehension of the product.

Methods of inquiry for the elliptic curve design
The second phase started with the investigation of possible elliptic curve configurations. Only binary field solutions are designed, because prime field calculations can be performed on current designs. Different high-level architectures for binary field ECC solutions are designed. A detailed comparison is made between the possible configurations. Gate count and performance were initially the main issues. The possibility to share hardware is investigated to reduce the area usage. Design reusable modules for different implementations. Countermeasures against power or timing attacks are taken. Looking at the power usage and timing sequences fake operations are added in the affine case. Using Montgomery in elliptic curve cryptography less time-consuming operations have to be performed, this increases the performance using projective coordinates.
The design is described in VHDL using EASE. The simulations are performed with Modelsim, the synthesis with Synopsys Design Compiler™.

**Results and conclusion towards the design of the elliptic curve cryptography system**

Designing the elliptic curve cryptography system, it turned out to be very competitive in comparison with current cryptography systems. Polynomial based implementations are recommended in comparison with normal based due to the scalability of the multiplier. The multiplier is designed to perform a multiplication, an addition and a shift-action. Two versions, affine and projective Montgomery coordinates, are implemented. Both designs have exchangeable modules. The basis calculations and the interfacing to the processor are similar for both configurations.

The elliptic curve solution uses less area and has better performance than the currently known cryptography systems. Therefore designing low-cost cryptography systems an elliptic curve implementation is recommended.
<table>
<thead>
<tr>
<th>Table of contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
</tr>
<tr>
<td>Table of contents</td>
</tr>
<tr>
<td>List of Figures, Tables and Algorithms</td>
</tr>
<tr>
<td>Literature</td>
</tr>
<tr>
<td>Abbreviations</td>
</tr>
<tr>
<td>Preface</td>
</tr>
<tr>
<td>Background of the company</td>
</tr>
<tr>
<td>Problem definition</td>
</tr>
<tr>
<td>Project definition</td>
</tr>
<tr>
<td>First phase</td>
</tr>
<tr>
<td>Second phase</td>
</tr>
<tr>
<td>1. Introduction</td>
</tr>
<tr>
<td>1.1 Reading instructions</td>
</tr>
<tr>
<td>1.2 Task partition</td>
</tr>
<tr>
<td>1.3 Cryptography in relation with the project</td>
</tr>
<tr>
<td>1.4 Public-key cryptography</td>
</tr>
<tr>
<td>1.5 Digital signature methods</td>
</tr>
<tr>
<td>1.6 Why using Montgomery multiplication</td>
</tr>
<tr>
<td>1.7 Prevent power and timing attacks that retrieve the key</td>
</tr>
<tr>
<td>2. Coding techniques in cryptography cores</td>
</tr>
<tr>
<td>2.1 Introduction</td>
</tr>
<tr>
<td>2.2 RSA</td>
</tr>
<tr>
<td>2.3 Elliptic curves</td>
</tr>
<tr>
<td>2.3.1 Introduction</td>
</tr>
<tr>
<td>2.3.2 Elliptic curve Parameters</td>
</tr>
<tr>
<td>2.3.3 Representations of field, coordinates and basis</td>
</tr>
<tr>
<td>2.3.4 Basic operations of elliptic curves</td>
</tr>
<tr>
<td>2.4 Elliptic curves in comparison with RSA</td>
</tr>
<tr>
<td>2.5 Montgomery Multiplication</td>
</tr>
<tr>
<td>2.6 Montgomery Exponentiation</td>
</tr>
<tr>
<td>2.6.1 Left-to-right binary exponentiation</td>
</tr>
<tr>
<td>2.6.2 Exponent Recoding</td>
</tr>
<tr>
<td>3. General info of the PIP2025 implementation</td>
</tr>
<tr>
<td>3.1 Introduction</td>
</tr>
<tr>
<td>3.2 PIP architecture</td>
</tr>
<tr>
<td>3.3 Bus interface</td>
</tr>
<tr>
<td>3.4 Hardware commands for the PIP via CDP</td>
</tr>
<tr>
<td>3.4.1 The communication between the processor and the PIP</td>
</tr>
<tr>
<td>3.4.2 Hardware commands</td>
</tr>
<tr>
<td>3.4.3 Control module that handles the coprocessor commands</td>
</tr>
<tr>
<td>3.5 Systolic array</td>
</tr>
<tr>
<td>3.6 The adapted processing element</td>
</tr>
<tr>
<td>3.7 How to perform a RSA calculation with the PIP</td>
</tr>
<tr>
<td>3.8 Performance and design restrictions</td>
</tr>
<tr>
<td>4. Software model of the PIP2025</td>
</tr>
<tr>
<td>4.1 Introduction</td>
</tr>
<tr>
<td>4.2 The design approach of the software model</td>
</tr>
<tr>
<td>4.3 Desired simulation level for internal and customer use</td>
</tr>
<tr>
<td>4.4 Class structure of the software model</td>
</tr>
<tr>
<td>4.5 Monitored values and variable declarations</td>
</tr>
</tbody>
</table>
4.6 Class functionality .................................................................................................................. 40
4.6.1 Bus interface ....................................................................................................................... 40
4.6.2 Modular Montgomery multiplication control (MMM-control) ........................................ 40
4.6.3 Exponentiation control (exp-control) and the parse digit algorithm ............................. 41
4.6.4 MMM class and its subclasses ............................................................................................ 41
4.7 User files .................................................................................................................................. 42
4.8 Testing of the model .............................................................................................................. 43
4.9 Performance ............................................................................................................................ 43
4.10 Perform tests using the software model .............................................................................. 44
5. Implement elliptic curves in a cryptography core ..................................................................... 45
5.1 Introduction ............................................................................................................................. 45
5.2 Architecture and performance issues .................................................................................... 45
5.3 Different bases and coordinate systems ............................................................................. 45
  5.3.1 Polynomial based with affine coordinates ...................................................................... 45
  5.3.2 Polynomial based with projective coordinates ................................................................. 46
  5.3.3 Normal based with affine coordinates ............................................................................. 46
  5.3.4 Normal based with projective coordinates ...................................................................... 47
5.4 Montgomery scalar multiplication (using projective coordinates) .................................... 47
5.5 Countermeasures against power and timing attacks ............................................................. 48
5.6 Comparison of different configurations .............................................................................. 48
5.7 Detailed comparison polynomial based ECC-implementations ........................................... 49
5.8 Other implementation issues ................................................................................................ 50
6. Design of binary field elliptic curve solution for a smartcard core ......................................... 53
6.1 Introduction ............................................................................................................................. 53
6.2 High-level architecture ......................................................................................................... 53
6.3 Basic polynomial based operations in binary field ............................................................... 54
  6.3.1 Polynomial addition and subtraction .............................................................................. 54
  6.3.2 Polynomial multiplication and squaring ......................................................................... 54
  6.3.3 Polynomial inversion ...................................................................................................... 55
6.4 Low-level architecture .......................................................................................................... 56
6.5 Area and performance estimations for the implemented ECC design .................................. 57
6.6 VHDL implementation of the elliptic curve design ............................................................... 58
  6.6.1 Similarity with the PIP2025 hardware ............................................................................ 58
  6.6.2 Control signals between the high-level modules ............................................................. 58
  6.6.3 Detailed timing diagrams for all possible situations ....................................................... 58
  6.6.4 Implementation of the (sub-)modules .............................................................................. 59
    6.6.4.1 The control module for both configurations ............................................................... 59
    6.6.4.2 The basic module ..................................................................................................... 60
    6.6.4.3 The execution module ............................................................................................. 60
    6.6.4.4 The register module for both configurations .......................................................... 60
    6.6.4.5 The interface module ............................................................................................. 61
6.7 Testing, simulation and synthesis .......................................................................................... 61
  6.7.1 Testing of the design ...................................................................................................... 61
  6.7.2 Simulation and syntheses results ..................................................................................... 62
Conclusions ..................................................................................................................................... 63
Conclusions towards the software model of the PIP2025 ............................................................ 63
Conclusions towards the binary field elliptic curve design .......................................................... 63
General conclusions towards the used strategies of the ECC design ........................................... 63
Recommendations .......................................................................................................................... 65
Recommendations for further development of the PIP2025 model and future models .................. 65
Recommendations for extension of the elliptic curve cryptography core ..................................... 65
Future cryptographic solutions for smartcard-implementations ................................................. 65
Appendices are combined in a separate document
List of Figures, Tables and Algorithms

List of Figures
Figure 2-1, A Modular exponentiation with Montgomery multiplications........................................... 22
Figure 3-1, High-level architecture of the cryptography core................................................................. 27
Figure 3-2, High-level PIP architecture .................................................................................................. 28
Figure 3-3, The old architecture of the exponentiation module............................................................... 32
Figure 3-4, The new architecture of the main PE (processing element).................................................. 33
Figure 4-1, High-level architecture of software model............................................................................ 39
Figure 4-2, Performance of the PIP, full exponentiation ........................................................................ 43
Figure 6-1, High-level architecture of the elliptic curve cryptography core.............................................. 53
Figure 6-2, Detailed architecture main modules elliptic curve implementation........................................ 56
Figure 6-3, Architecture of the matrix shift-and-add multiplier (depth of matrix is two)......................... 57
Figure 6-4, Final architecture of the multiplier (width is M, depth is 1).................................................... 60
Figure 6-5, Comparison of expected and actual performance figures and gate count................................. 62

List of Tables
Table 2-1, Comparison of RSA and ECC key-sizes [15]........................................................................ 21
Table 3-1, The registers of the PIP........................................................................................................... 29
Table 3-2, Number of MMM-operations for a complete exponentiation (worst case)............................. 34
Table 3-3, Number of cycles for one MMM-operation ............................................................................ 34
Table 4-1, Simulation Level of the program............................................................................................ 38
Table 4-2, Hierarchical levels class structure.......................................................................................... 38
Table 4-3, Program file instructions........................................................................................................ 42
Table 4-4, Expected number of cycles specified for a full worst case 1024-bit exponentiation................ 44
Table 5-1, Comparison of different bases ............................................................................................... 48
Table 5-2, Comparison of different coordinates systems ......................................................................... 49
Table 5-3, Performance figures polynomial bases elliptic curve implementations................................. 50

List of Algorithms
Algorithm 2-1, Scalar multiplication with example.................................................................................. 21
Algorithm 2-2, Montgomery left-to-right binary exponentiation [2]......................................................... 23
Algorithm 5-1, Montgomery scalar multiplication with example............................................................. 47
Algorithm 6-1, Polynomial addition in a binary field................................................................................ 54
Algorithm 6-2, Polynomial multiplication in a binary field...................................................................... 55
Algorithm 6-3, Polynomial inversion in a binary field (based on the euclidean algorithm)...................... 55
Literature


[17] Xu S.B., "ECC Hardware Architecture, Feasibility Study – Multiplier Architectures over Binary Fields", September 28th 2001 (internal)


[22] Cuppens W., G. Willemse, "System Specification PIPECC – Modular Arithmetic Accelerator", April 2002 (internal, accessory of this project)
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBA</td>
<td>Advanced Micro-controller Bus Architecture</td>
</tr>
<tr>
<td>ARM</td>
<td>Advanced RISC Machine</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>CID</td>
<td>Caller Identity Delivery</td>
</tr>
<tr>
<td>CIDCW</td>
<td>Caller Identity Delivery on Call Waiting</td>
</tr>
<tr>
<td>CRES</td>
<td>Crypto and embedded software group</td>
</tr>
<tr>
<td>CRT</td>
<td>Chinese Remainder Theorem</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>DEA</td>
<td>Data Encryption Algorithm</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital Signature Algorithm</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic Curves Cryptography</td>
</tr>
<tr>
<td>ECDSA</td>
<td>Elliptic Curves Digital Signature Algorithm.</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standards</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GF</td>
<td>Abbreviation of a Field representation in elliptic curves – Galois Field</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>LNAU</td>
<td>Large Number Arrhythmic Unit</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instructions Per Second</td>
</tr>
<tr>
<td>MME</td>
<td>Montgomery Multiplier Engine</td>
</tr>
<tr>
<td>MMM</td>
<td>Montgomery Modular Multiplication</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute for Standards and Technology</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PIP</td>
<td>Pijnenburg Intellectual Property</td>
</tr>
<tr>
<td>PKCS</td>
<td>Public Key Cryptography Systems standards</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest, Shamir and Adleman (the names of the ‘inventors’ of the RSA-algorithm)</td>
</tr>
<tr>
<td>SA</td>
<td>Systolic Array (or Security Association)</td>
</tr>
<tr>
<td>SAFER</td>
<td>Secure And Fast Encryption Routine</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>SHS</td>
<td>Secure Hash Standard</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
</tbody>
</table>

---

Implementing Cryptographic Solutions in a Modular SmartCard core

Copyright © SafeNet. All rights reserved.
Preface

The graduation project has taken place at SafeNet B.V. (former: Pijnenburg Securealink B.V.). The project started in September 2001 and ended in June 2002. H. Janssen and C. Jansen coordinated the activities at SafeNet. Prof. M.P.J. Stevens and A. Verschueren from the Computer, Networks and Design chair supervised the project at the Technical University of Eindhoven. Close cooperation was required with the System Architecture Group and the Crypto Research Group of SafeNet BV.

Background of the company

SafeNet is a global market-leader within the highly specialized and emerging network security market. SafeNet merged at January 2nd 2002 with Pijnenburg Securealink BV (PBSL), the company where the project started. Due to the merger the name, Pijnenburg Securealink BV changed in SafeNet. SafeNet’s mission statement, similar to the PBSL missions statement, is to be the leading provider of Security Solutions in Silicon™. Their products offer the most advanced and effective security-solutions in silicon available in the market. The company’s security chips offer low-cost and low-power dissipation implementation of encryption algorithms in combination with highest levels of performance and tamper resistance. This particular combination: complex encryption engines, embedded processor and security features, all offered as one-chip solutions, is unique in its market. It combines dedicated cryptographic engines with the possibility to run security software applications on chip. It provides a secure environment, which software-only security applications running on an open platform cannot offer. The security features are among the most advanced available in silicon today.

Problem definition

The project will study, specify, define and implement a low-end cryptographic IP core. The core will be optimal for a smartcard implementation. The design is therefore low power and has as less gates as possible without losing too much of performance. In order to provide building blocks for low-end cryptography systems, the project encounters several technological and other challenges:

• Currently used techniques and implementations of the exponentiation algorithm are not suitable for the new demands with respect to performance, small area, small memory usage and low power consumption.
• The current implementation uses an embedded proprietary 16-bit RISC processor. The solution must be able to support other popular 32-bit embedded processors like the ARM7 or MIPS3000/4000; both on chip bus interfaces and embedded software is affected.
• New protocols and algorithms like elliptic curves using modular arithmetic have been defined; the solution need to support these algorithms effectively.
• The solution will be embedded into future chip products of SafeNet as well as Intellectual Property for third parties. This requires a different approach with respect to reusability, documentation, technology independence, available software libraries and hardware support.

The solution will be proven using one of the existing FPGA prototyping platforms available at SafeNet. When not suitable a new platform can be developed in this project.

Project definition

The low-end cryptographic IP core that has to be designed is called the PIP2025 (PIP). The PIP should execute coprocessor instructions that cannot be handled by the RISC processor efficiently.
The project is divided in two phases. In the first phase the emphasis will be on the RSA implementation. In the second phase a solution for elliptic curves should be designed.

**First phase**

The first phase is divided into 5 steps that tackle the problem and deliver a high performance software model of the PIP coprocessor. A SafeNet's modular exponentiation core.

1. Learn and collect information about two algorithms: RSA and elliptic curve and their mathematical operations.
   - Modular exponentiations.
   - Modular multiplications.
   - Modular additions.
2. Learn, Inventory of the currently available models within SafeNet.
   - RSA-EXP models.
   - Elliptic curve prime field model (GF(p)).
   - Elliptic curve binary field (GF(2^m)) probably available in December (TU/e).
3. Define and build 'C' behavioral model for the PIP, which allows:
   - Early software development without the PIP hardware available.
   - Generation of intermediate results for hardware debugging assistance.
   - Customer design integration model.
4. Characterize performance for RSA modular exponentiations with and without CRT.
   - Check if performance of the model is according to specification
   - Define next possible optimization options.
5. Study and compare performance of the behavioral 'C' model with the FPGA model.
   - Investigate the correlation.

**Second phase**

Implementation of the binary field elliptic curves algorithm GF(2^m) and do recommendations to develop a combined design with the RSA-solution, the PIP.

First of all is chosen which binary field algorithm is used for the implementation. Therefore different algorithms have to be compared on the following issues:

- Maximum clock speed and number of cycles.
- Number of gates.
- Power dissipation.
- Memory usage (memory is very expansive in the smartcard business).

Determine which algorithm performs the best on the three issues. Afterwards an implementation can be made of the algorithm. Choose one of the following options as basis for the second phase.

1. Take the TU/e-SafeNet project from two foreign students, a non-commercial GF(2^m) implementation, and commercialize the project taking into account issues such as:
   - gate count (<35k).
   - power dissipation (~10mW).
   - ARM interfacing.
   - development of VHDL test-bench.
   - An 160-bit ECC-calculation in less than 100 msec. at 13MHz.
2. Design a small cryptography-core that handles RSA, GF(p) and GF($2^m$) less than 60k gates and limited RAM usage. This can be divided in multiple steps:
   (a) Investigate implementation of GF(p) in the systolic array of the PIP.
   (b) Implement GF(p) in the PIP (partly in software).
   (c) Combine the PIP and the design from step 1 into one core.
3. Design a new small cryptography-core that handles elliptic curve GF($2^m$) less than 40k gates and limited RAM usage. This can be divided in multiple steps:
   (a) Investigate implementation of GF($2^m$) of the necessary calculations.
   (b) Optimize and compare hardware solutions of all binary field configurations
   (c) Implement GF($2^m$) in a hardware core
   (d) Investigate a combined solution of both hardware implementations
1. Introduction

1.1 Reading instructions
The thesis consists of four parts. The first part, chapter one and two, introduces cryptography. After a global introduction, the two most important algorithms are discussed in detail. Chapters three and four cover the first part of the project. In these chapters the development and the functionality of the software model is explained. The third part involves the designing of a future cryptographic system for a low-cost implementation. The development of the design and simulations of the implementation are discussed in chapter five and six. Finally the thesis contains the conclusions and recommendations of both project phases. The appendices are combined in a separate document: “Implementing Cryptographic Solutions in a Modular SmartCard core – Appendices”.

1.2 Task partition
At first we both studied the cryptographic algorithms, RSA and elliptic curves. Then the important implementations for our project available at SafeNet are investigated in detail. All possible implementations for elliptic curve are studied and summarized.
Starting with the software model, we divided the tasks. Willem developed the user interface and the communication between processor, memory and the coprocessor. Extra options, such as different simulations levels and trace-files are appended. The systolic architecture and the control unit are designed by Gijs. The systolic architecture is improved in comparison with the available hardware design. These improvements are taken over in the new IP-product.
During the second phase Willem studied the normal based configurations and Gijs the polynomial based configurations, both within the binary field. The choice which configuration is designed is made during discussions. Willem developed the point-adding and point-doubling for both implementations. The scalar multiplication for both configurations and the basic operations are developed by Gijs. The final hardware implementation is done by both of us. Willem implemented control and interface modules. Gijs implemented the basic, execution and register modules.

1.3 Cryptography in relation with the project
The need for security of data is growing day by day. This is mainly caused by the enormous growth of the digital communication via public networks like Internet and mobile phone networks.
Cryptography is one of the most important ways to secure this data. SafeNet examines the cryptographic standards in order to use this knowledge to develop and produce integrated solutions for data security. Currently cryptographic hardware solutions are mainly used in very expensive server environments such as m-commerce, m-banking, micro payments. On the other hand a secured smartcard requires strong cryptographic implementations that are low-cost, low-power and flexible.
The current scaleable private key cryptographic solutions that SafeNet can offer are extremely powerful and competitive in the high-end part of the market. Feasibility study at SafeNet has revealed that the technology can be adapted towards the low-end area. This project studied, specified, defined and implemented a low-end cryptographic core.

1.4 Public-key cryptography
Many cryptography-systems have been developed to secure data transmissions. The systems require that both sender and receiver possess the cryption-key. Therefore this key must be transferred over a
safe channel. The save key-transfer can be avoided using a public-key crypton algorithm. Today’s most common algorithm is RSA. In this and other public-key systems, both receiver and sender have their own public and private key. The encrypted message with a public key can only be decrypted with the private key [3].

A disadvantage of RSA and many other algorithms, is the necessary key-size to have secure communication\(^1\). At the moment a key-size of 1024 bits will provide a secure message. Although in the near future a key-size of 2048 bits is desired. A new algorithm is elliptic curve cryptography; the key length for the same security level is factors smaller in comparison to other algorithms. Elliptic curve will be investigated and discussed in Paragraph 2.3.

1.5 Digital signature methods

Currently there are two digital signature methods used for authentication between two parties that want to communicate. These methods are RSA signatures and DSA signatures. The RSA algorithm is used within the developed PIP-core and is discussed in the first part of this thesis. A new third algorithm introduced in the second part of the thesis is ECDSA. ECDSA signatures are smaller than RSA and ECDSA keys require less bandwidth than RSA keys. Furthermore, on many platforms ECDSA operations can be computed faster than similar strength DSA and RSA operations.

In chapter 2 the different coding techniques used in this project will be discussed in detail. These are RSA, ECC and an introduction in calculations in the Montgomery domain. In the next paragraph the advantage of performing calculations in the Montgomery domain is explained.

1.6 Why using Montgomery multiplication

Montgomery multiplication is a solution for a modular multiplication without a slow modular operation. Convert the modulus into a multiplication and the division ends with shifting. This converts a difficult modular operation into an easy modular operation.

To use Montgomery, all the data has to be transformed to the Montgomery domain, the calculation is executed and then all the data is re-transformed (converted to a normal representation). The large numbers are transformed and divided into multiple parts. Per part the multiplication is executed, although the modulus is taken into account immediately. When all parts of both operands are multiplied with each other adding and shifting the sub-results of the multiplications can derive the final result of the multiplication. To speed up the calculation the sub-results that are available will immediately added to the final result parallel to the other multiplications. The mathematics of Montgomery will be discussed in Paragraph 2.5.

1.7 Prevent power and timing attacks that retrieve the key

Power and timing attacks are observations of differences in power dissipation and execution sequences. With these differences it is possible to reveal the key or message. To make it difficult to reveal the key or message we have to look at the power and timing during the calculations. Briefly this means the total and fragment execution time and power dissipation should not depend on the message and or key. Otherwise it could be possible to derive the key from multiple test vectors. Therefore power and timing attacks should be made senseless on the core.

\(^1\) Under ‘secure communication’ is meant that the encrypted messages can not be decrypted without the key within considerable time, using current (computer) techniques

---

Copyright © SafeNet. All rights reserved.

Page 18
2. Coding techniques in cryptography cores

2.1 Introduction

For safe data transmission it is necessary to encrypt data. Many different encryption algorithms can be used for the encryption. Two algorithms that are proven to be very secure are RSA (Rivest, Shamir and Adleman, the names of the ‘inventors’) and elliptic curves. In the following chapters the use of both algorithms will be explained. The Montgomery Multiplication and Montgomery Exponentiation that can be used to speed-up the encryption will be discussed after the cryptographic algorithms.

2.2 RSA

RSA is an encryption-algorithm that is often used in many security solutions. To encrypt and decrypt messages this algorithm uses two keys, the public- and private-key. With one of those keys the other cannot be derived. RSA encryption is based on two large prime numbers P and Q chosen by the receiver. With these two numbers, the numbers N and \( \varphi \) are created by (an example can be found in Appendix A):

\[
N = P \times Q \quad \text{Equation 2-1}
\]

\[
\varphi = (P-1) \times (Q-1) \quad \text{Equation 2-2}
\]

The next step is to choose random a number E, which has to be a relatively prime number. Therefore the greatest common divider of \( \varphi \) and E has to be 1. In other words \( \varphi \) cannot be divided by E. The numbers E and N are public and therefore will be sent to the sender. After finding a number E, the number D for the receiver has to be calculated by:

\[
D = \frac{1}{E} \mod \varphi \quad \text{Equation 2-3}
\]

\[
E \times D = 1 \mod \varphi \quad \text{Equation 2-4}
\]

The number D can be found with the Euclidean algorithm (an example can be found in Appendix A). With these three numbers, messages will be encrypted by the sender and decrypted by the receiver. For instance a message M will be encrypted into an encrypted message C by:

\[
C = M^E \mod N \quad \text{Equation 2-5}
\]

The receiver will decrypt the encrypted message C:

\[
M = C^D \mod N \quad \text{Equation 2-6}
\]

At this moment the length of the public keys is in order of 1024 bits. The developed coprocessor design has to handle keys with a maximum length of 2048 bits.

2.3 Elliptic curves

2.3.1 Introduction

Elliptic curve cryptography systems are based on an elliptic curve group, that is a set of points on an elliptic curve with a point at infinity. Mathematicians defined a point-adding operation in the elliptic curve group, which possesses the algebraic properties of ordinary addition (e.g., commutative and associative). Elliptic curve points can be added but not multiplied. It is, however, possible to perform scalar multiplication, which is another name for repeated addition of the same point. If we have an
integer $n$ and a point $P$ on an elliptic curve, we can simply compute $Q = nP$ (scalar multiplication) by adding $n$ copies of $P$. However, it is very hard to compute $n$ even if we know $P$ and $Q$ and all parameters of the elliptic curve. This is the elliptic curve discrete logarithm (ECDL) problem.

2.3.2 Elliptic curve Parameters

In each elliptic curve cryptography system, both communicators have to agree on a set of parameters, called EC domain parameters. These parameters include a field $GF(q)$, where $q$ is a positive odd prime integer $p$ or $2^m$ for some positive integer $m$. Two elliptic curve coefficients $a$ and $b$, elements of $GF(q)$, that define an elliptic curve $E$. A positive prime integer $r$ dividing the number of points on $E$ and a curve point $G$ of order $r$ ($G$ is called the generator of a subgroup of order $r$). Depending on protocols used, users might need to generate EC domain parameters. In IEEE P1363 Annex A, Chapter A.12 [9] and SEC 1 of Certicom Corp.© [10] have described the algorithms for generating and validating EC domain parameters. In WAP, however, WTLS has already recommended nine sets of EC domain parameters. Thus, communicators don't need to generate their own set of domain parameters in the applications of WAP; the recommended domain parameters are summarized by Certicom Corp.© [11].

Once EC domain parameters have been chosen for an elliptic curve cryptography system, each party can choose a random number $s$ from $[1..r]$ as his private key and compute $W = sG$ as his public key[12], [15].

2.3.3 Representations of field, coordinates and basis

The main operation in the two possible fields (prime field $- GF(p)$ and binary field $- GF(2^m)$) is the scalar multiplication mentioned above. The scalar multiplication contains multiple point-addings and point-doublings, dependent on the binary representation of the scalars.

The total number of bits of the scalar is similar to the number of point-doublings. The Hamming-weight of the scalar ($W_H(k)$ : number of ones in the binary representation) represents the number of point-addings. These two point-operations consist subsequently of multiple operations, dependent on the coordinate system. In the class-diagrams of Appendix N are the different possibilities summarized. The two coordinate systems that can be used are affine and projective. Due to the extra coordinate in the projective representation, the number of inversions can be decreased in exchange for other operations. The sequences of the calculations of the four different representations are summarized in Appendix O.

Except the field and coordinates, in binary field also the basis of the coordinates has to be chosen: "what will the coordinates represent". Two representations are useful, this are normal and polynomial basis. The large numbers represent a string with an element of the elliptic group to the power $2^i$ or a polynomial.

Normal basis:

$$a_0 \cdot \theta + a_1 \cdot \theta^2 + a_2 \cdot \theta^{2^2} + ... + a_{m-1} \cdot \theta^{2^{m-1}}$$

$a_i \in \{0, 1\} - \theta \in GF(2^m)$

Polynomial basis:

$$a_{m-1} \cdot x^{m-1} + ... + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

$a_i \in \{0, 1\}$

Implementing an elliptic curve cryptography system a field, basis and coordinate system has to be chosen [14]. Which representations are chosen to be implemented in the core is discussed and argued in Chapter 5.
2.3.4 Basic operations of elliptic curves

As mentioned in previous paragraph the basic operation is the scalar multiplication. This multiplication involves multiple point-addings and point-doublings. In the sequence below a simple four-bit scalar is multiplied with point P. The bit-representation of the scalar assigns the number of point-doublings and -addings. The pseudo-code for the scalar multiplication can be found in Algorithm 2-1.

Algorithm 2-1, Scalar multiplication with example

```
//CALCULATE Q = s · P
P => (Xp, Yp)
\ s => (Sm, ..., S1, S0), with Sm = 1
Q => (Xq, Yq)

Q = P;
for (int i = m-1; i >= 0; --i)
  \ Q = 2Q;
  if (Si == 1)
    Q = Q + P;
```

Example: scalar s = 13
\Rightarrow 13 = 1101_2
\Rightarrow 1_2 \rightarrow Q = P
\Rightarrow 1_2 \rightarrow Q = 2Q + P = 3P
\Rightarrow 0_2 \rightarrow Q = 2Q = 6P
\Rightarrow 1_2 \rightarrow Q = 2Q + P = 13P
Q = 13P = (2(2(2P + P)) + P)

Dependent on the chosen representation, field and coordinate system, a point-adding or -doubling consists of a sequence of basic operations (addition, multiplication and inversion). The sequences for binary and prime field are summarized in Appendix O. In case of binary field two different bases for the coordinates have to be distinguished. These different bases do not influence the calculation sequence, but involve the basic operations. How these operations are performed will be discussed later. An example of important high-level elliptic curve calculations is shown in Appendix B.

2.4 Elliptic curves in comparison with RSA

Currently elliptic curves deliver the highest strength per bit of any known public-key cryptosystem. This strength is based on the ECDLP-problem [16] and means smaller key-sizes to yield equivalent levels of security. Table 2-1 compares the key-sizes needed for equivalent strength security in ECC with RSA and DSA. Given the best-known algorithms to factor integers and compute elliptic curve logarithms, the key-sizes are considered to be equivalent strength based on MIPS years needed to recover one key.

<table>
<thead>
<tr>
<th>Time to break in MIPS years [16]</th>
<th>RSA/DSA key-size</th>
<th>ECC key-size</th>
<th>RSA/ECC key-size ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^4$</td>
<td>512</td>
<td>106</td>
<td>5 : 1</td>
</tr>
<tr>
<td>$10^8$</td>
<td>768</td>
<td>132</td>
<td>6 : 1</td>
</tr>
<tr>
<td>$10^{11}$</td>
<td>1024</td>
<td>160</td>
<td>7 : 1</td>
</tr>
<tr>
<td>$10^{20}$</td>
<td>2048</td>
<td>210</td>
<td>10 : 1</td>
</tr>
<tr>
<td>$10^{78}$</td>
<td>21000</td>
<td>600</td>
<td>35 : 1</td>
</tr>
</tbody>
</table>

Table 2-1, Comparison of RSA and ECC key-sizes [15]
2.5 Montgomery Multiplication

In 1985 Peter Montgomery published a new method for modular multiplication that avoided the trial division used in other methods [1]. This trial division was a relatively time consuming operation. A short introduction to this method is given here. More details can be found in [2]. The conditions for a Montgomery Multiplication are shown in Appendix C.

At the abstract level a Modular Montgomery Multiplication (MMM) is the following:

\[ \text{MMM}(X,Y) = X \cdot Y \cdot R^{-1} \mod N \quad \text{Equation 2-7} \]

Where R is chosen such that division by R is easy; usually R is chosen to be a power of 2. R and N should be relatively prime. Using this MMM(X,Y) function for modular multiplication, a transformation and a back transformation are needed. The steps are shown in the next figure.

![Figure 2-1, A Modular exponentiation with Montgomery multiplications](image)

In general, this MMM(X, Y) function is calculated in the following way:

\[ m_m = (X \cdot Y \mod R) \cdot N' \mod R \]
\[ t = (XY + m_m \cdot N) / R \]
\[ \text{if } (t < N) \text{ then return } t \]
\[ \text{else return } t - N \]

Where N' is such that \( RR^{-1} = NN' = 1 \). A more comprehensive example can be found in Appendix C. Essential is that \( t \) is an integer. The proof of this can be found in literature [3] and [4].

Normally the Modular Multiplication calculated in the Montgomery domain consists of four Montgomery Multiplications. First the two operands are transformed to the Montgomery domain. Then the actual multiplication is performed followed by the back transformation of the result. These four Montgomery Multiplications are reduced to two Montgomery Multiplications and a reduction of the result. This are one transformation and the calculation of the result, if the result is larger than N, subtract N from the result. Both the described sequences are shown in Figure 2-1, the reduction is explained in the next example.
\( T_{\text{Res}} := \text{Mont}(A, R^2 \mod N) \)
\( = A \cdot (R^2 \mod N) \cdot R^{-1} \mod N = A \cdot R \mod N \)

**"Transformation"**

\( C := \text{Mont}(B, T_{\text{Res}}) \)
\( = B \cdot (A \cdot R \mod N) \cdot R^{-1} \mod N = B \cdot A \mod N \)

**"Multiplication"**

if \( C > N \) then \( C = C - N \)

**"Reduction"**

In the original paper of Montgomery it is required that \( R > N \). The Montgomery multiplication can be divided in separate smaller multiply reduction steps. These reduction steps can be mapped onto a systolic array implementation. The systolic array consists of \( P \) processing elements with word size \( a \) and \( b \). The processing elements multiply words from the operands (X and Y), add a temporary result and add multiple times the corresponding N-word. The temporary result, the N-word and the Y-word are \( b \)-bits width; the size of the X-word is \( a \)-bit. R is chosen on the next conditions (the operand that is feed horizontal to the systolic array has size \( a \)):

\[
\rho = \left\lfloor \frac{n + a + 2}{(a \cdot P)} \right\rfloor \\
r = \rho \cdot a \cdot P - a \\
R = 2^r
\]

**2.6 Montgomery Exponentiation**

The Montgomery Exponentiation is implemented as a repeated Montgomery Multiplication. The exponentiation is based on two principles: left-to-right binary exponentiation and exponent parsing. Two initial values have to be calculated: A and \( x' \). The processor calculates the required \( R^2 \mod N \) using software calculations. To reduce the number of multiplications needed for one exponentiation the exponent can be re-coded. We use exponent parsing for the re-coding of the exponent.

**2.6.1 Left-to-right binary exponentiation**

The pseudo-code for left-to-right binary exponentiation is shown in Algorithm 2-2.

```
Algorithm 2-2, Montgomery left-to-right binary exponentiation [2]
// Calculate \( A = g^e \mod N \)
g \in \mathbb{G} \) with \( (g < N), e = (e_m, e_{m-1}, ..., e_1, e_0)_2 \)
A = g;
for(int i = m-1; i >= 0; --i)
{
    A = 2 \cdot A;
    if \( (e_i == 1) \)
        A = g \cdot A;
}
```

An exponentiation consists of \((m-1)\)-squarings. If \( W_H(e) \) is the Hemming weight (number of ones) of the exponent then another \( W_H(e) \) multiplications are performed. The total number of multiplications for one exponentiation is thus \( m + W_H(e) \), with \( W_H(e) \) the Hemming weight of \( e \).

In example 1 of Appendix D, Montgomery Multiplication and Left-to-right Exponentiation are combined to calculate a modular exponentiation.
2.6.2 Exponent Recoding

Exponent Recoding can be used to reduce the number of multiplications for an exponentiation at the cost of some pre-calculations. In the PIP Exponent Recoding is chosen. In [2] this principle can be found under String-replacement and Sliding Window exponentiation. In general the “window size” is the maximum length of the string that can be replaced. It determines the number of pre-calculations also.

Exponent re-coding, “window size” 4, with a maximum exponent of 7:

\[
\text{exponent } e = 0xEA629593:
\begin{align*}
1110 & \quad 1010 \quad 0110 \quad 0010 \quad 1001 \quad 0101 \quad 1001 \quad 0011 \\
\text{is recoded into:} & \\
0070 & \quad 0005 \quad 0003 \quad 0000 \quad 0005 \quad 0000 \quad 0005 \quad 0003 \quad 0001 \quad 0003
\end{align*}
\]

For the calculation of \( A^e \mod N \) the following odd powers of \( A \) must be pre-calculated:

\( A^1, A^3, A^5, A^7 \)

In the PIP the number of memory locations restricts the number of pre-calculations. In example 2 of Appendix D an example of an exponentiation can be found.

The reduction in cycles using exponent parsing can be mathematically verified [8], for one extra odd power the example is given below.

When the exponent contains the sequence ‘11’, the corresponding calculations can be reduced by 25%. Combining the two exponents just three instead of four MMM-operations are necessary to calculate the following temporary result.

This gives the following reductions for the bit combinations of the exponent (the chance of appearing of the sequence is given in percentage at the end of the line):

- ‘00’ - no reduction (2 MMM-operations) 25%
- ‘01’ - possible reduction (x instead of 3 MMM-operations) 25%
- ‘10’ - no reduction (3 MMM-operations) 25%
- ‘11’ - reduction (3 instead of 4 MMM-operations) 25%

In case of ‘01’, the next bits from the value are important, dependent on the next bits \( x \) can be calculated. Here are the possibilities summarized and afterwards \( x \) is calculated.

- ‘0’ - no reduction: \( x = 3 \) 50%
- ‘10’ - reduction: \( x = 2 \) 25%
- ‘110’ - no extra reduction: \( x = 3 \) 12.5%
- ‘1110’ - reduction: \( x = 2 \) 6.25%
- ‘11110’ - no extra reduction: \( x = 3 \) etc.

This continues and gives a Taylor chain \( y = (25\% + 6.25\% + \ldots + 100·{1 \over 2}^{2n}\% + \ldots) \) that summarizes the percentage of situations in which \( x = 2 \). In all the other situations obtains \( x = 3 \).

The average value of \( x \) is then two times \( y \) and three times \( 1 - y \), this results in:

\[
x = 2\cdot y + 3\cdot(1-y) = 3 - y = 3 - {1 \over 4} - {1 \over 16} - \ldots - {1 \over 2^{2n}} - \ldots \quad \text{Equation 2-8}
\]

Copyright © SafeNet. All rights reserved.
In Equation 2-9 the overall reduction with one odd power is calculated. The used and calculated values are from an average case situation, where the exponent consists of an equal number of zeros and ones.

\[
\text{From: } \frac{1}{4} [2 + 3 + 3 + 4] = 3 MMM^{3/2 \text{bits}}
\]

\[
\text{To: } \frac{1}{4} \left[ 2 + \left( 3 - \sum_{n=1}^{\infty} \left( \frac{1}{2} \right)^{2n} \right) + 3 + 3 \right] = \frac{1}{4} \left[ 11 - \frac{1}{3} \right] = 2 \frac{2}{3} MMM^{3/2 \text{bits}} \\
\text{Equation 2-9}
\]

\[
\text{Reduction: } \frac{3 - 2 \frac{2}{3}}{3} \times 100\% = 11.1\%
\]

With using infinite number of bits the reduction is about 11.1\%, because the used values are large (1024 or more bits), the reduction will be more than 11\%.
3. General info of the PIP2025 implementation

3.1 Introduction

The PIP2025 coprocessor (PIP) is a cryptography accelerator core. It operates together with a processor, this is 32-bit RISC-based processor for system-on-silicon devices. Hardware coprocessor commands for the coprocessor number of the PIP will be decoded and executed by the PIP. The PIP is able to control the memory and databus in case of a data processing instruction, this in contrast to a standard coprocessor. When the coprocessor has no access to the bus, the PIP is in a wait-state. It is triggered by the coprocessor signals of the processor to the PIP and the hardware instructions on the databus. In the figure below the highest-level architecture is shown.

![Figure 3-1, High-level architecture of the cryptography core](image)

3.2 PIP architecture

The PIP is divided in three main parts, the control, the bus interface and the MMM. The separate FIFO and N-Register are a part of the MMM, but they are directly accessible by the processor. In Figure 3-2 the top-level of the PIP architecture is visible.

The control unit is divided in two control parts: the MMM control and the EXP control. The control will let the MMM-unit execute the requested calculations by using the control-signals to the MMM-unit. The data that is used during a calculation is stored in several registers (i.e. X, FIFO and N). The MMM is in fact the exponentiation accelerator where the PIP is designed for. It consists of multiple Processing Elements (further called PE) and two termination PE's. The MMM has registers to store the temporary results and counter values from the calculations. The bus interface is the typical interface for a 32-bit RISC-processor. The registers in the interface will be directly accessible by the ARM with coprocessor instructions. The control unit uses the data from these registers during the calculations. The MMM control unit directly triggers the MMM module to execute a cycle with the provided data. The data is requested by the control unit and passes the interface. For an exponentiation (EXP command) the control unit has to execute some extra functionality to reduce the number of MMM calculations (the parse digit algorithm - Paragraph 2.6.2).
3.3 Bus interface

The PIP contains an interface based on the ARM® SC100 bus interface. Each instruction that is intended for the coprocessor will be executed by the PIP. The PIP decides to execute the instruction on behave of the nCPI signal and the coprocessor-number in the instruction. The PIP will execute the instruction when the coprocessor number is valid. The PIP uses five coprocessor instructions. These are:

1. CDP – Coprocessor Data Processing with opcode (command):
   - Reset
   - MMM
   - MMMNext
   - EXP
2. LDC – Load Coprocessor
3. MCR – Move to Coprocessor from ARM Register
4. MRC – Move to ARM Register from Coprocessor
5. STC – Store Coprocessor

The instructions are bit-wise specified in Appendix E

The CDP instruction triggers the coprocessor to execute the included command. Meanwhile the processor is in a wait-state. The commands will be discussed in the next paragraph. The MCR instruction moves the data to a PIP-register from a processor-register. With the bits 16-19 of the instruction can be selected which register of the processor will be copied to the PIP-register.
The MRC instruction moves the data to a processor-register from a PIP-register. The following registers are accessible with MCR (W) and MRC (R) instructions.

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Access</th>
<th>Size</th>
<th>Register Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n_acc</td>
<td>R/W</td>
<td>8</td>
<td>Least significant bits of N' in Montgomery.</td>
</tr>
<tr>
<td>1</td>
<td>b_pointer</td>
<td>R/W</td>
<td>asize*</td>
<td>Memory address for exponent (decrements during operation)</td>
</tr>
<tr>
<td>2</td>
<td>b_counter</td>
<td>R/W</td>
<td>11</td>
<td>Number of bits to go in exponent (decrements during operation)</td>
</tr>
<tr>
<td>3</td>
<td>x_base</td>
<td>R/W</td>
<td>asize*</td>
<td>Base address of the 'odd powers of X' array.</td>
</tr>
<tr>
<td>4</td>
<td>y_base</td>
<td>R/W</td>
<td>asize*</td>
<td>Base address of the result and Y operand.</td>
</tr>
<tr>
<td>5</td>
<td>n_base</td>
<td>R/W</td>
<td>asize*</td>
<td>Base address of the N operand, only use if N size &gt; 1024 bits.</td>
</tr>
<tr>
<td>7</td>
<td>n_ydigits</td>
<td>R/W</td>
<td>7</td>
<td>Number of 32-bit words in Y operand.</td>
</tr>
<tr>
<td>8</td>
<td>n_xdigits</td>
<td>R/W</td>
<td>7</td>
<td>Number of 32-bit words in X operand.</td>
</tr>
<tr>
<td>9</td>
<td>n_passes</td>
<td>R/W</td>
<td>9</td>
<td>Number of passes over N/Y for each multiplication.</td>
</tr>
<tr>
<td>10</td>
<td>n_exparray</td>
<td>R/W</td>
<td>4</td>
<td>Number of elements in 'odd powers of X' array.</td>
</tr>
<tr>
<td>11</td>
<td>n_mode</td>
<td>R/W</td>
<td>1</td>
<td>Mode of the N operand.</td>
</tr>
<tr>
<td>12</td>
<td>status</td>
<td>R</td>
<td>2</td>
<td>Busy or overflow status bits.</td>
</tr>
</tbody>
</table>

Table 3-1, The registers of the PIP

*Note: In Table 3-1 asize specifies the size of an address. The two least significant bits (LSB) of each address-register are always zero. The address-registers have only as many bits as useful to access the RAM.

The LDC instruction provides data for the coprocessor N-register directly from the RAM. The processor provides the addresses of the data on the addressbus. The n_ydigits-register contains the number of N-words that are needed. Each cycle the processor provides a new address on the addressbus so the memory can load data from the RAM onto the databus. Then the PIP reads the data from the databus and stores it into the N-register.
The STC instruction stores the result from the coprocessor N-register directly to the RAM. The processor provides the addresses of the data on the addressbus. Each cycle the PIP sets the data on the databus and the processor provides an address on the addressbus to let the memory store the data from the databus into the RAM.

3.4 Hardware commands for the PIP via CDP

3.4.1 The communication between the processor and the PIP

The RISC-processor reads instructions from the RAM. Because the processor uses a coprocessor to accelerate multiplications, it is possible the processor reads instructions that cannot be handled by the processor itself. The processor and the PIP have an interface that exchange commands. The processor triggers the PIP with the nCPI-signal, the PIP answers with the CPA (coprocessor acknowledge) and CPB (coprocessor busy) signals. In the beginning the processor is the master of the bus, but when the PIP is busy the processor is halted. In case of an interrupt the PIP is set in a 'wait'-state that makes it possible to continue the process where it was interrupted. The registers are directly accessible by the processor. The FIFO is used internal by the PIP, the temporary results are stored in it.

Examples of instruction timetable from the processor to the PIP and from the PIP to the RAM are shown in Appendix F.
3.4.2 Hardware commands

- **Reset**
The reset command resets registers and counters. After a reset, a new operation can be started.

- **MMM (Y = X·Y mod N)**
This command performs a modular Montgomery multiplication. It multiplies parts from the operands parallel and adds the values to the temporary result. The final result is stored in the RAM.

For the MMM hardware function the following operands are necessary:

- X-operand
- Y-operand
- N-operand: in register when N ≤ 1024 bits, in RAM when N > 1024 bits

The result will be written on the place of the Y-operand in the RAM.

- **MMMNext (Z = X·Y mod N)**
The MMMNext hardware command is similar to the MMM command, only the result will be written in the RAM after the Y-operand instead of over the Y-operand.

- **EXP (X = X^B·R^1 mod N)**
This command performs a full multiplication in the Montgomery domain. During the calculations all values (except the exponent) are in this domain. This means X has to be transformed to the Montgomery domain on beforehand, transformation can be done by executing an MMM-command with X and R^2 as operands.

The exponentiation consists of multiple MMM-operations, dependent of the exponent. To reduce the number of MMM calculations odd powers of X are stored in the memory. A register holds the number of odd powers. At first x^2 is calculated because x and x^2 are the basis for the other odd-powers. When this number is four the following odd-powers are available in the memory: x, x^3, x^5 and x^7; x^2 will be rewritten by the temporary results of the exponentiation. The EXP calculation is prepared in software, the combinations of one’s in the exponent are recognized in hardware, parallel to the MMM-operations.

When all the bits from the exponent are handled, the EXP-command is finished. But the result is still in the Montgomery domain. Therefore it is to be re-transformed by an MMM-command. The operands of this MMM-operation are the temporary result and ‘1’.

Executing the command all the large number registers and RAM values are read from LSB to MSB, mostly in pairs of multiple bits. The exponent B is read from MSB to LSB when the EXP-command is executed. R^2 is dependent on N and calculated on beforehand. It is partly calculated in software and partly with an EXP-command in the PIP.

3.4.3 Control module that handles the coprocessor commands

The control module handles the commands included with the CDP instruction from the processor to the PIP. The four possible commands are a Reset, an MMM (Modular Montgomery Multiply), an MMMNext and an EXP (modular exponentiation) operation.

The control module controls the execution of one of these commands, the EXP operation is a repeated MMM operation. The necessary values are stored in the registers. The operations are discussed per command in this chapter.

**Reset command**
The reset command resets the counters used in the control module. This means X_pointer, Y_counter and B_counter. Other important values such as n_passes and B_pointer have to be reloaded before the next command. The registers that have to be reset are Y_reg, N_reg and R_reg. Resetting these registers performs also a reset through the systolic array.

**MMM command**

The first thing done by the control module, executing an MMM command, is checking the command register. If this register is zero, the number of bits of N is smaller or equal to 1024 and the N-register will be used to store N. If the command register is set, the number of bits of N is larger than 1024. In that case the N-register is used as FIFO and the temporary result will be stored in it. The FIFO-in-pointer is set on n_ydigits to generate ‘R = 0’ for the complete first pass of Y. Then the MMM calculation is started. The calculation consists of a counter that controls the loading of X parts. Two recursive for-loops, one that handles the passes and the loading of X-parts and one inner loop that controls a pass, loading of y-part and triggering of the SA. When the last Y-part in the last pass is loaded, extra cycles are needed to complete the calculation. Parallel with these last cycles the result is stored over the Y-operand in the RAM, until the complete result is stored.

The timing of this MMM command is shown in Appendix F. The RAM memory addresses of the X-operand and Y-operand are endowed with the MMM-function, these values are loaded from respectively T-base and Y-base.

The fetched registers of the PIP remain the same during the MMM operation. T-base added with X-pointer points to the next part of X that has to be loaded in the X-register. Similar is the value that points to the next Y determined by adding Y-base with Y-pointer. Y-pointer is reset every pass, X-pointer after a complete MMM-operation.

**MMMNExt command**

The execution of this command is similar to the MMM command. After executing an MMMNExt command the result is stored after the Y-operand in the RAM, instead of over the Y-operand.

**EXP command**

The main part of the execution of an EXP command is done by repeated MMM operations. To decrease the number of MMM operations, bits from the exponent can be merged to one MMM operation. This is exponent re-coding and is performed by the parse digit algorithm treated in Paragraph 2.6.2. Therefore odd powers of X have to be calculated with pre-calculations. When the pre-calculations are executed, the actual EXP operation is started. The exponent is scanned per bit. After every 32 bit of the exponent a new exponent-part is loaded. Is a bit zero, one MMM operations is necessary to calculate the quadrate. The MMM function is called with the Y base address on both address fields. When a bit from the exponent is one, two MMM operations are necessary to calculate the next temporary result. Combining multiple ones to another odd power of X reduces the number of MMM operations. In case of a one, first an MMM operation with Y base on both fields is executed, this means squaring of Y. Followed by the execution of an MMM operation with the address of the odd power of X and Y base on the address fields, this is a multiplication of the both operands. The results of MMM operations are stored over the Y-operand in the RAM.

**3.5 Systolic array**

The systolic array handles an MMM operation, although the control module controls the MMM command. SafeNet has developed an architecture that accelerates MMM operations. This architecture
consists of multiple processing elements (PE) that parallel multiply two parts of both operands. Per multiplication the modulus is taken into account. Therefore the temporary result will never be larger than the modulus.

In the architecture the idea is that all parts from one multiplicand are multiplied by all parts of the other multiplicand. Shifting and adding the temporary results delivers the final result of the MMM operation. The number of bits from the different multiplicands is usually not the same. The sizes of the parts for the multiplication could also be different per multiplicand. In the PIP one multiplicand is divided in α-bit parts, the other in β-bit parts. The PIP consists of multiple PE’s and two termination PE’s. These two blocks are always necessary, irrespectively of the number of multiplication blocks. The termination PE’s recover the result from three independent values. The final architecture of the systolic array in the PIP is shown in the Appendix H.

The input values are X, N and Y. The 1024 or 2048 bit values X and Y are provided by the RAM memory, N is stored locally in advance or in the RAM, dependent on the configuration. The final result is \( x \cdot y \mod N \). The temporary result is stored in a FIFO and used in the next pass through the systolic array at the input side.

In the figure is immediately visible that a lot of temporary signal are generated in the PE’s these signals are overflows and delta-corrections of the multiplication in a PE. The termination PE combines the temporary value, a carry from the previous result and a delta-correction into one 32-bit result.

![Diagram](image)

**Figure 3-3, The old architecture of the exponentiation module**

### 3.6 The adapted processing element

The processing elements in previous chips of SafeNet contain the exponentiation module from Figure 3-3. In this module X and Y are multiplied and added with previous temporary values. To take in account the modules, for a certain X, C times N has to be added to the result, where C is a constant value for that specific X. For further explanation of the old PE [6].

In this module all the calculated output-values, except the doxor value, are transported to the next PE. The doxor is transported to a following PE. The longest path starts with an alpha \( \times \) beta bit.
multiplication of X and Y, followed by a \((\beta+1) \times (\alpha+\beta) \times (\alpha+\beta)\) bit addition. Then an alpha-bit value is diverted from the result and transported to the following PE. Finally an alpha \times alpha bit multiplication is calculated and stored in a register. Parallel to the first multiplication two additions are performed. The delay due to the multiplication is larger than for the additions.

To reduce the number of registers and avoid the passing of doxor: a new architecture is designed. In this architecture the registers are reduced with 40 bit (~280 gates in TSMC 0.18) per PE for current configuration. Both termination PE's can be merged, because the doxor of the last PE is transported to the first termination PE instead of the second. The longest path is also decreased. The number of operations is the same, but the large addition involves only two operands instead of three. The path involves a multiplication followed by an addition and finally a small multiplication. Parallel with both multiplications an addition is performed. In this case the multiplications cause more delay than the addition. The new architecture of the main PE is shown in Figure 3-4, the architecture of all the PE's is shown in Appendix I.

![Figure 3-4, The new architecture of the main PE (processing element)](image)

### 3.7 How to perform a RSA calculation with the PIP

In Paragraph 2.2 is explained that large number modular exponentiations are necessary to perform secure encryption of data with RSA. For encoding as well as decoding of the message a large number modular exponentiation is desired (Equation 2-5 and 2-6). The PIP can be used to perform the en- and decoding of the message. But using the PIP the base operand of the exponentiation has to be in the Montgomery domain. Therefore the operand has to be transformed. This can be done with \(R^2\) (Paragraph 2.5), this value will be calculated on beforehand, using the CPU and the PIP.

After calculating \(R^2\) the registers from the PIP are loaded with data, this is done with the coprocessor instructions, discussed in Paragraph 3.3. The registers that need to be loaded can be found in Table 3-1. Afterwards the base operand from the exponentiation is transformed to the Montgomery domain, this operation is executed by an MMM-command. When the parse digit algorithm is used to accelerate the exponentiation the odd powers have to be calculated. This involves extra instructions.
for reloading registers and one MMM-command for every extra odd power. Finally the instruction with the EXP-command is fetched and triggers the PIP to perform the exponentiation. The result is still in the Montgomery domain, to generate the en- or decoded message, the result need to be re-transformed. This is done with a CDP-instruction that performs an MMM-command. The operands are the temporary result and ‘1’ (Paragraph 2.5). The result of the last MMM-operation is the encoded or decoded RSA message.

3.8 Performance and design restrictions

The PIP should be able to handle a full 1024-bit exponentiation in 95ms at 20MHz according to current specification. This means the worst case scenario has a maximum of 1.9M cycles for a full 1024-bit exponentiation. The worst case scenario occurs when the exponent consists of only ones. The following table shows how many multiplications are necessary with or without the use of extras odd powers (Paragraph 2.6.2).

<table>
<thead>
<tr>
<th>#odd powers \ # bits</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (x)</td>
<td>1026</td>
<td>2050</td>
<td>4098</td>
</tr>
<tr>
<td>2 (x, x^3)</td>
<td>771</td>
<td>1539</td>
<td>3075</td>
</tr>
</tbody>
</table>

Table 3-2, Number of MMM-operations for a complete exponentiation (worst case)

Using the mentioned specification, one MMM-operation should take about 925 cycles (1.9M / 2050) without the extra odd power. With the use of the first extra odd power ~1230 cycles (1.9M / 1539) are allowed for one MMM-operation.

To satisfy the performance, the systolic array has to satisfy the following equation.

\[
\begin{align*}
\text{n}_\text{passes} & = \left( \frac{\text{#bits}}{\beta} + 1 \right) \times \text{cycles} \quad \text{for cycles see Table 3-3} \\
\text{n}_\text{passes} & = \left( \frac{1024 + \alpha}{\text{#PE’s} \times \alpha} \right) \\
\end{align*}
\]

Equation 3-1

Equation 3-2

The four variables (#bits, #PE’s, α and β) are essential for the number of cycles of an MMM-operation.

<table>
<thead>
<tr>
<th>alpha 16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PE’s 8</td>
<td>8</td>
</tr>
<tr>
<td>4 33*65 = 2145</td>
<td>33*33 = 1089</td>
</tr>
<tr>
<td>8 17*65 = 1105</td>
<td>17*33 = 561</td>
</tr>
<tr>
<td>12 11*65 = 715</td>
<td>11*33 = 363</td>
</tr>
</tbody>
</table>

Table 3-3, Number of cycles for one MMM-operation

Using one extra odd power, the two smallest configurations that satisfy Equation 3-1 are 8 x 8 x 16 and 4 x 8 x 32 (#PE’s x α x β). The chosen architecture from Paragraph 3.5 is one of these configurations. Without extra odd powers, α, β (max 32, because of the bus-width) or the number of PE’s need to be increased to satisfy the performance requirement. In the model that will be designed, odd powers will be used to accelerate the exponentiation. Using odd-powers, the configuration with α = 8, β = 32 and #PE’s = 4 satisfies the specification. Due to the smallest number of gates this configuration is used to develop the cryptography core. With this configuration one MMM-instruction
takes about 1100 cycles. A worst case exponentiation is therefore handled in 85ms (at 20MHz). This execution time satisfies the specification.

There are other configurations possible, but because the total gate count must be small, every larger design is less desired. Therefore the smallest possible design that satisfies the specification will be chosen. The choice of technology is important. In comparison with the registers, the multipliers are smaller in TSMC 0.18 than in TSMC 0.25. For larger values of $\alpha$ and $\beta$ the multipliers become more significant, therefore the differences between both technologies become greater for larger $\alpha$ and $\beta$.

From this moment all the numbers and percentages will refer to the TSMC 0.18 technology.

In the actual design every extra gate takes extra silicon and is therefore more expensive. In general doubling the number of PE's, doubles the number of gates of the systolic array. Doubling $\alpha$ doubles the number of gates for the X, Gxor and Nacc registers and the corresponding multipliers. This is approximately 75% of a PE. Doubling $\beta$ will double the number of gates of the other registers and corresponding multipliers and adders. This results in approximately 90% of a PE. These results show that $\alpha$ and $\beta$ should be as large as possible. Although due to the bus-width $\beta$ is restricted to 32-bit. Therefore $\alpha$ has to be smaller or equal to 16 bit. An extra adder has to be added to the PE in the 16x32-bit configuration. The extra adder takes extra gates, a 8x32-bit solution is the best option. Four PE's are necessary to satisfy the specification in the 8x32-bit solution.
4. Software model of the PIP2025

4.1 Introduction

The developed software model has three goals. It should be able to test the developed software libraries of SafeNet with the software model. This makes it possible the software is tested and optimized without the actual design is ready. The second function of the model is providing potential customers a model of the PIP without giving them the complete design. Finally the model should give insight into the operation of the system, from high-level to bit-level. How the model is designed and which steps are taken during the development of the model will be discussed in the next chapter. The comparison with the hardware implementation should be made every step in the development of the software model. In the software model timing problems should be tackled. Issues like power dissipation, number of registers and memory use will be investigated. After the development, an estimation of these values will be given with simulations and calculations using the model.

4.2 The design approach of the software model

The followed approach developing the model consists of the next steps:

- Brainstorm about the highest and lowest hierarchical level
- Design a class model
- Implement the high-level classes
- Check the high-level functionality and the communication between the classes
- If necessary adapt the model and implement the lower-level classes
- Check the complete functionality of the system and the correctness of the results

During the brainstorm session a general architecture is designed. At the moment of this session the complete specification of the PIP was not available. Afterwards a basic design of the necessary classes is developed. For simulating coprocessor instructions a processor class and an interface class are designed. These classes make it possible to put and get instructions respectively on and from the databus. The instructions can be checked on bit level on both windows. To test the communication between the processor and the PIP, a basic simulation function is realized. For simulating a clock-cycle a Clock-class is triggered when all the calculations from one cycle are handled. A Trace class is developed to track all the desired temporary and final results from the simulation. To store the traced values the trace-class generates a file. Except the basic simulation-function that exists in the SmartCrd-class, it has to be possible to simulate the complete coprocessor instruction set (Paragraph 3.3). These instructions are loaded on the highest level. This level contains the high-level bus-structures and the processor, PIP and Memory. The processor- and Memory-classes can only be monitored on the highest level. On the other hand, during the execution of these instructions it is possible to step into the PIP and trace the coprocessor instruction handling. On all the levels, from system down to bit-level, registers and bit wise operations can be monitored.

At this phase a coprocessor instruction is decoded and the corresponding command is read. Dependent on the command (Paragraph 3.4), the interface-class handles the interaction on the address- and databus. The implementation of the functionality from the PIP is started. One of the instructions makes the coprocessor act completely independent. The PIP executes the command and meanwhile it controls the databus. Except the classes that control the execution of the command, a separate class is developed that contains the registers and the FIFO. All these components are used to store temporary data and pointers to the memory.
Finally all the classes are implemented with their complete functionality. These are all the necessary communications with other classes, all the calculations and the right timing in comparison with the hardware implementation. In the following paragraphs the complete development of the model will be explained. Important design issues will be discussed in detail.

### 4.3 Desired simulation level for internal and customer use

The software model is build for internal use, but can be used by customers. Therefore several simulation levels are possible. The most abstract version is level 0, the lowest-level version is level 3. The different levels are shown in a Table 4-1, more about the differences between the simulation levels can be read in Appendix K.

<table>
<thead>
<tr>
<th>Simulation Level</th>
<th>useful options</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, customer</td>
<td>Simulate a standard program. A trace file can be created and high-level classes can be monitored during the simulation.</td>
</tr>
<tr>
<td>1, customer</td>
<td>Similar to level 0 but simulation of several program files with their memory files is possible. It allows the user to create program and memory files. The executed instructions are shown in a Trace window.</td>
</tr>
<tr>
<td>2, internal</td>
<td>Similar to level 1 but all the windows of the PIP except the systolic array are accessible. Frequency, $\alpha$, $\beta$ and the number of PE’s are variable</td>
</tr>
<tr>
<td>3, internal</td>
<td>Similar to level 2, but on this level the PIPMMM-window is accessible for the four-PE configuration. Each instruction can be monitored on bit level</td>
</tr>
</tbody>
</table>

Table 4-1, Simulation Level of the program

The reason of dividing the possibilities for internal users and customers is to prevent the customer to re-engineer the SafeNet’s cryptography core. A customer who is just interested in the product is shown or given the level ‘0’ model. When the customer becomes a potential buyer and wants to simulate more in detail, the level ‘1’ model can be provided. This will probably get along with the signing of a first phase contract. For internal use the program is divided into two levels, for the engineers a simulation on bit-level. For all the other employees a higher level model that does not show the lowest level channels and the content of the registers.

### 4.4 Class structure of the software model

The first step developing the software model is translation of the product specification into a class model. This class model contains inheritance, associations and aggregations. This explains the hierarchical structure and the relations between the classes. Separating the inheritance from the other relations holds the model easy to interpretate. This results in two separate class models that are shown in Appendix G. In these models the hierarchical levels from Table 4-2 can be distinguished.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>the simulation</td>
</tr>
<tr>
<td>B</td>
<td>the complete smartcard</td>
</tr>
<tr>
<td>C</td>
<td>the CPU, the PIP and the memory</td>
</tr>
<tr>
<td>D</td>
<td>high level modules of the PIP</td>
</tr>
<tr>
<td>E</td>
<td>sub-modules of the high-level PIP modules</td>
</tr>
<tr>
<td>F</td>
<td>functions that operate on bit-level</td>
</tr>
</tbody>
</table>

Table 4-2, Hierarchical levels class structure
In the model the first three levels (A, B and C) are immediately visible, this is called simulation level 0. If desired the other levels can be accessed during the simulation. The communication and data exchange between the three main modules of the smartcard simulation are always visible (Figure 4-1). The communication between the high-level modules occurs with the addressbus, the databus and the control signals. In the figure are the highest level classes and the communication channels shown. Per level such a window is developed to visualize the architecture.

![Figure 4-1, High-level architecture of software model](image)

The memory class has a restricted size. It is able to receive and return data according to the endowed address. The processor of the smartcard (CPU) handles the instructions. In the model this class generates the coprocessor instructions, these are the software instructions that can not be handled by the processor. The third process class on level C is the coprocessor itself. This class has to be designed down to bit-level and is therefore divided into sub-classes. Furthermore some classes are needed to control and trace the simulation. Hierarchical level D (similar to simulation level 1) contains the main modules of the PIP. One class controls the communication, another class controls the execution of a coprocessor command. The functionality (MMM) and the main registers of the coprocessor are divided in two separate classes. These registers are directly accessible by the processor with the different coprocessor commands. The other registers and counters are located in the classes itself. The control class of the coprocessor is mainly used executing the coprocessor commands. Therefore for the two main commands separate control classes are designed. Of course the functionality of the MMM has to be simplified in multiple classes. The lowest level of the class structure (level E) contains the processing elements and the detailed control modules (Appendix G).

Clicking with the mouse on the modules/boxes it is possible to make the contents of the modules visible. A new window appears with its internal architecture. In this way it is possible to step through
every hierarchical level of the model. The lowest level shows the registers, the systolic array and if desired the values on the channels. This level is similar to simulation level 3. The remaining windows are shown in Appendix J.

4.5 Monitored values and variable declarations

The monitored values differ dependent on the simulation level (Paragraph 4.3). The result of a simulation is put in a trace file and can be read with a text-editor. In this paragraph there will be explained what can be monitored at level 0. After that, there is explained for each next level what can be visualized in comparison with the previous levels.

With the level '0'-model in the trace file the next signals can be shown: nCPI, CPA, CPB, nMRequest, nRW, Total, Clock, the instruction that will be fetched, decoded and executed. The values of the Data- and Addressbus can be monitored. At the end of a simulation the actual simulation time is determined. With the configurable frequency the execution time for a real-time hardware implementation is determined. Both values can be found in the trace file. The result of the calculation is traced in the trace-file when this is set in the program file.

With the level '1'-model the registers that are loaded or stored with a MCR or MRC instruction are traced and stored in the trace file. Each command of a CDP instruction is shown. During the calculation of an MMM-, MMMNext- or EXP-command, the loading of the values from the memory is traced and stored in the file (these values are words of X, R, N or Y).

With the level '2'-model all the values of the XReg[PE] in each PE is shown and the values of the N, R and Y that are loaded/stored in the register or FIFO. The stored words in the N-register during a LDC or STC instruction are shown in the trace-file. Executing an EXP-command the current MMM-operation is indicated and the corresponding exponent-bit number are shown in the trace-file.

With the level '3'-model all registers of each PE and channels between the PE's are shown. The new values of X and mi are shown.

For internal use there are two versions available. The configuration where $\beta = 32$ and a configuration where $\beta = 16$. For both configurations $\alpha$, the number of PE's and the frequency are adaptable. The coprocessor number is adaptable but this won't influence the simulation. The user can change the instruction (and the command) in the CPU window to visualize the instruction in bits.

4.6 Class functionality

The complete functionality of all the classes, the methods and the attributes are described in a separate document [7]. In this paragraph the functionality of the most important classes will be shortly described.

4.6.1 Bus interface

The interface class decodes the instruction that is available on the Databus. The coprocessor checks whether the instruction is for the PIP. Then the interface checks the significant bits from the instruction and prepares the PIPControl-class for execution of the instruction. It passes the addresses and the data from and to the PIP during the execution of the instruction.

4.6.2 Modular Montgomery multiplication control (MMM-control)

The MMM-control involves the loading of X, Y, N and R. Most important are the triggering/resetting of the registers, the passing of the right addresses to the memory and the updating of the counters.
Every pass an X-word will be loaded, for every PE eight x-bits are available. After loading the X-word all Y-words are loaded parallel with the corresponding N-words (in case of N <= 1024 bits). These words are passed through the systolic array. After the first pass, a temporary result is sent from the FIFO to the systolic array. When the first Y-part is through the systolic array, the first temporary result can be stored. When the last pass is finished, the final result becomes available. If all the Y-parts are loaded this result can be stored, parallel to the execution of the last cycles in the systolic array. When the execution is finished and the last results are stored, a new command can be executed. Every pass has to be checked whether there are results or Y-words available. The initialization of the PE's takes place after the beginning of a new pass. The first PE is immediately initialized, the next cycle the second PE, until all PE's are initialized. This has to be done every pass. During the initialization a new X-value is loaded and the mi-register is triggered.

The discussed case obtains a N smaller or equal to 1024 bits. Although the control module handles N values up to 2048 bits. In the remaining cases the situation is similar except that the N-word is loaded from the memory before the Y-word. During the inserted cycles for the loading of N, the systolic array is in a wait-state. Finally the MMM control handles the MMM commands and the MMMNext commands. Executing the MMMNext commands the result addresses are increased with the number of Y-digits plus one.

### 4.6.3 Exponentiation control (exp-control) and the parse digit algorithm

Receiving an EXP-command the exp-control module of the PIP is triggered. This module determines, according to the bits from the exponent, which sequence of MMM operations is necessary to calculate the result of the exponentiation. The exp-control loads b-words and starts the 'parse digit'-algorithm (Paragraph 2.6.2) parallel to the first MMM. According to the result of this algorithm the next multiplication is executed. The EXP-control uses the MMM-control, discussed in the previous paragraph, to calculate the MMM's.

Timing and power attacks (Paragraph 5.5) are made senseless in the implemented 'parse digit'-algorithm (Paragraph 2.6.2). Therefore before every MMM operation a B word is loaded. Four bits of the B-word are stored, because the window for one exponent parse is four bits. In total eight bits from the exponent will be stored, because the four bits that need to be inspected can distributed over two words.

### 4.6.4 MMM class and its subclasses

The MMM class processes the received data. It gets data from the memory and the global and local registers. After triggering the MMM it passes the data to the systolic array and when necessary stores the result in the FIFO. The class generates the initialization signals for the registers in the PE's. The MMM is divided into the actual systolic array and a small control module. The main subclass is the systolic array (Appendix H), in this array the Montgomery calculations takes place. Dependent on the number of PE's, alpha and beta (Paragraph 3.6) the two multiplication operands are shifted through the array. The lowest level classes are the subclasses of the systolic array. These are the processing elements itself. Four different elements can be distinguished. The first element and last element differ from the main processing element and will therefore be separate classes. To correct the previous beta-bit result with the delayed alpha-bit delta value from the last element a termination element is necessary.
4.7 User files

With our software model it is possible to use files for loading a program and initializing the memory. The memory contains variables for the calculations. A program file contains multiple instructions. All commands and values on one line in a command-file have to be separated by a TAB-character. An example of a program file is shown in Appendix J.4, a memory file is shown in Appendix J.5 and an example of a trace file is shown in Appendix J.6. In Table 4-3 the instructions for the program file are explained. The instructions in the program-file are case sensitive.

<table>
<thead>
<tr>
<th>Instruction line from the file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCR name* 0xHHHHHHHH</td>
<td>MCR instruction with Register number name*, Register data 0xHHHHHHHH</td>
</tr>
<tr>
<td>CDP MMM</td>
<td>CDP instruction with &quot;MMM&quot;-command</td>
</tr>
<tr>
<td>CDP MMMNext</td>
<td>CDP instruction with &quot;MMMNExt&quot;-command</td>
</tr>
<tr>
<td>CDP EXP</td>
<td>CDP instruction with &quot;EXP&quot;-command</td>
</tr>
<tr>
<td>CDP Reset</td>
<td>CDP instruction with &quot;Reset&quot;-command</td>
</tr>
<tr>
<td>LDC 0xHHHHHHHH</td>
<td>LDC instruction with basic-address 0xHHHHHHHH</td>
</tr>
<tr>
<td>STC 0xHHHHHHHH</td>
<td>STC instruction with basic-address 0xHHHHHHHH</td>
</tr>
<tr>
<td>MRC name*</td>
<td>MRC instruction with Register number name*</td>
</tr>
<tr>
<td>MEM Load memory.mem</td>
<td>The memory file that will be loaded</td>
</tr>
<tr>
<td>MEM Result 0xHHHHHHHH</td>
<td>The place of the result in the memory</td>
</tr>
<tr>
<td>MEM Stored store.mem</td>
<td>The memory file that will be stored</td>
</tr>
<tr>
<td>MOV src dst</td>
<td>Move the data from the src (0xHHHHHHHH) to the dst (0xHHHHHHHH) in the memory of the smartcard</td>
</tr>
<tr>
<td>SDC On</td>
<td>Show Data Changes on</td>
</tr>
<tr>
<td>SDC ShowMessage</td>
<td>Messageboxes are possible</td>
</tr>
<tr>
<td>SDC HideMessage</td>
<td>Messageboxes are not possible</td>
</tr>
<tr>
<td>SDC Off</td>
<td>Show Data Changes off</td>
</tr>
<tr>
<td>TRC Open trace.trc</td>
<td>Open a new trace file, works only after another open trace-file was closed</td>
</tr>
<tr>
<td>TRC On</td>
<td>Trace-file on</td>
</tr>
<tr>
<td>TRC Off</td>
<td>No tracing</td>
</tr>
<tr>
<td>TRC Close</td>
<td>Close an open trace file, when one was used</td>
</tr>
</tbody>
</table>

Table 4-3, Program file instructions

* Note: name is defined as: "N_acc", "B_pointer", "B_counter", "T_base", "Y_base", "N_base", "X_pointer", "n_ydigits", "n_xdigits", "n_passes", "n_exparray", "n_mode" or "status"

The memory file consists per line 2 variables the first is an address and the next is the data that will be stored on the address in the memory of the smartcard.

With these options it is possible to create several trace-files for debugging. Placing TRC-instructions before and after the coprocessor instructions make it possible to start and stop tracing during the simulation. While running it is possible to show the data changes when the SDC-instructions are used. After the simulation of the program it is possible to create a memory-file with the data that is stored in the memory of the smartcard. This is possible to put an instruction MEM store and a filename on the last line of the program. When a program is started and no memory is loaded this will be detected by
the model, if desired a memory-file can instantly be loaded. When no program is opened, the model asks to open a program-file or to run the standard program.

### 4.8 Testing of the model

For testing of the model, SafeNet's internal test vectors for large number multiplications and exponentiations are used. This are values for A, B and N (Paragraph 2.5 and 2.6) from one up to 2048-bit, a command and the result of execution of this command with these parameters. To generate \( R^2 \) (Paragraph 3.7) another internal software model is used. Because a memory file is necessary to initialize the RAM memory a file has to be generated according to the test vector. Therefore a conversion program is developed that converts a test-vector to a useful memory-file. In the program-file (Paragraph 4.7) the desired instructions can be written. After loading the memory the instructions are executed. The result can be read from the memory-file. If desired the final result can be compared with a preloaded value from the memory. The software model gives the right results for all the tested vectors. In case of an exponentiation there is one restriction: the exponent has to be larger than one, because the exponent pointer initially points to the second bit of the exponent.

### 4.9 Performance

The performance of the software model, introduced in Paragraph 3.7, will be further discussed in this paragraph. First the worst case scenario is executed for multiple lengths of N and different lengths of the exponent. The averages are determined mathematically. All the values are determined with and without the use of extra odd powers. In Appendix L the number of cycles for a complete exponentiation are shown; the important values are bold. A complete exponentiation contains a transformation, calculation of the odd powers, the exponentiation and the retransformation of the result. In Figure 4-2 different scenarios for a complete exponentiation without odd-powers are shown. Generally doubling the number of bits increases the number of cycles eight times. Although when 1024-bit is exceeded or in case of another configuration this factor could be larger.

![Figure 4-2, Performance of the PIP, full exponentiation](image)

In the worst case scenario there is no difference between two and three odd powers, because the exponent consists of only ones in the worst case situation. The combination of ‘101’ (5) does not occur in the exponent and the third odd power \((x^5)\) is never used. All thought the difference between one and two odd powers is almost a 25% reduction (Figure L-0-1). This can be explained by the
'parse digit'-algorithm (Paragraph 2.6.2). The complete exponent consists of ones, the second odd power can be used to reduce the number of MMM-operations with 25% for the exponentiation. The average values show difference performance for the three odd powers (Figure L-0-2). In case of three odd powers the exponentiation is reduced with about 16%, in case of two odd powers the exponentiation is reduced with 11%. These values can be mathematically verified, this is shown in Paragraph 2.6.2. The results are similar to the expected values. The expected value for a full 1024x1024-bit worst case exponentiation is calculated in Table 4-4. This number equals the number of cycles of the actual simulation.

<table>
<thead>
<tr>
<th>A 1024-bit MMM-operation takes:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loading first value</td>
<td>3 cycles</td>
</tr>
<tr>
<td>33 passes with loading of one x-part and 33 y-parts</td>
<td>$33*(1+33) = 1122$ cycles</td>
</tr>
<tr>
<td>finish calculation</td>
<td>5 cycles</td>
</tr>
<tr>
<td>storing of the result</td>
<td>33 cycles</td>
</tr>
<tr>
<td>finish MMM</td>
<td>1 cycle</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>1164</strong> cycles</td>
</tr>
</tbody>
</table>

For the 1024-bit worst case exponentiation counts:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>2 cycles</td>
</tr>
<tr>
<td>14 MCR instructions</td>
<td>$15*3 = 45$ cycles</td>
</tr>
<tr>
<td>one LDC instruction</td>
<td>34 cycles</td>
</tr>
<tr>
<td>one MOV instruction (copy 32 words to another addresses)</td>
<td>2 cycles $^2$</td>
</tr>
<tr>
<td>two MMM instructions (transformation and retransformation)</td>
<td>$2*1164 = 2328$ cycles</td>
</tr>
<tr>
<td>initialize MMM instructions</td>
<td>$2*1 = 2$ cycles</td>
</tr>
<tr>
<td>one 1024-bit EXP instruction</td>
<td>load first B-word</td>
</tr>
<tr>
<td>1023 MMM(Y, Y)</td>
<td>$1023*1164 = 1190772$ cycles</td>
</tr>
<tr>
<td>1023 MMM(X, Y)</td>
<td>$1023*1164 = 1190772$ cycles</td>
</tr>
<tr>
<td>load every MMM a B-word</td>
<td>$2*1023 = 2046$ cycles</td>
</tr>
<tr>
<td>finish EXP</td>
<td>1 cycle</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>2386007</strong> cycles</td>
</tr>
</tbody>
</table>

Table 4-4, Expected number of cycles specified for a full worst case 1024-bit exponentiation

4.10 Perform tests using the software model

The software model can be used to generate input files for the VHDL-model. It is possible to generate a VHDL command-file that provides all the inputs and outputs for a module or a set of modules. Finally a command-file is generated that is used to test the complete functionality of the PIP. In Appendix M a part of a VHDL command-file is shown. This file generates the inputs and checks the outputs of the systolic array from the PIP. Except the outputs, internal values of the systolic array can be shown in the file, these values are pure for the verification of the single processing elements. The software model is used to give customers insight into the PIP implementation. Furthermore the model can be used to generate intermediate and complete input vectors and results for hardware debugging.

$^2$ The MOV-instruction takes more cycles than mentioned, but is performed by the ARM and is therefore not taken into account in this model.
5. Implement elliptic curves in a cryptography core

5.1 Introduction
Before an elliptic curve cryptography core can be designed an elliptic representation has to be chosen. The different representations are already discussed in Paragraph 2.3.3. Dependent on the desired gate count and performance, field, basis and coordinate system have to be chosen. In this implementation only the binary field is interesting. Prime field calculations can be performed on the current PIP with its RISC-processor. Modular multiplications are performed by the PIP, the other operations by the ALU of the RISC-processor. This is possible because the operations are performed with normal large numbers. In the binary field implementations the operations that need to be performed do not involve integers, but binary representations of other sequences. Therefore operation-related hardware has to be designed to perform a fast binary field calculation. The elliptic curve implementation involves a low-cost, large-scale IP-product, therefore the area usage is the most important issue. Of course, the performance has to be competitive. In this and the next chapter the size of the field will be called $M$ or $m$. When a value has size $M$ or $m$ it means that the value has the same number of bits as the reduction polynomial that belongs to a $2^m$-field.

5.2 Architecture and performance issues
To perform a scalar multiplication, multiple point-addings and point-doublings have to be calculated. These point-operations consist of multiple standard operations within a certain basis. The operations that have to be performed are additions, multiplications and inversions of binary values. In polynomial based configurations the values represent polynomials; in a normal based configurations the values represent a number in the $2^m$-field. For each operation is investigated which basis elements are needed to calculate the result. For every operation is examined whether the implementation could be combined with other necessary hardware to reduce the number of elements. The expected number of gates and the number of cycles are compared for the different designs. These numbers involve a complete scalar multiplication. In the next paragraphs the expected number of gates and the performance of the different designs are discussed. In the next chapter the implementations of the best configurations are described and a detailed implementation is shown. For the four discussed configurations, brainstorm sessions delivered several high-level designs that should perform a binary field elliptic scalar multiplication. For the most interesting designs a detailed comparison is made in paragraph 5.7.

5.3 Different bases and coordinate systems
It is clear that only binary field implementations are interesting for this project due to the current market demands and because prime field calculations are possible on current designs. In this paragraph the advantages and disadvantages of the four mentioned implementations (Paragraph 2.3.3 and Appendix N) are discussed.

5.3.1 Polynomial based with affine coordinates
The scalability of a polynomial based implementation with affine coordinates is large. In the implementation, gates are reserved for a polynomial-adder, -multiplier and -inverter. The addition similarly used for a subtraction consists of only XOR-ports. The number of ports is equal to the number of bits. The multiplier can be designed using a matrix of shift-and-add multipliers. The basic
principles of this multiplier can be found in [17] and [18] and are discussed further in the next chapter. The depth of the matrix is reverse proportional with the number of cycles of a multiplication. To prevent bypassing, the depth should be a factor of the number of bits. The inversion consists of multiple shift- and add-operations [19].

Except the logic that performs the actual operations a control-part is necessary that triggers the execution-blocks. The control unit consists of a scalar, doubling- and adding-module. The last two modules are triggered dependent on the bit-sequence of the scalar.

The number of temporary registers is scalable with a minimum of three m-bit registers and eight addresses pointed to the coordinates and polynomial in the memory. When the number of memory accesses should be reduced, five m-bit registers are desirable. These are four temporary registers and one register for the reduction polynomial. Several temporary values can be stored locally and accessed immediately. The other values are stored in the RAM and loaded when they are needed. When enough local registers are available, values can be loaded during the previous operations. The polynomial is stored in the register unit and directly connected to the multiplication unit. In the SEC2 recommendations [11] the recommended polynomials are shown. Most of the bits of the polynomial are always zero, therefore not the complete polynomial has to be stored.

Finally the interface module has to be configured, this module can be included from the PIP. The interface that includes memory access and decoding instructions is similar, only the triggering of the control-unit and the registers has to be adapted.

5.3.2 Polynomial based with projective coordinates

The other polynomial based implementation is with projective coordinates. Projective coordinates introduce an extra coordinate to decrease the number of divisions. Inversions are time consuming operations. Before and after a complete calculation the coordinates have to be transformed. Transformation involves one inversion, back-transformation involves a multiplication. This configuration reduces the number of inversions, but increases the number of multiplications. Therefore the inversion could be performed in software. This saves gates, but decreases safety and performance.

The number of operations that need to be performed is different, but the execution of the operations is the same due to the same basis. Therefore performance differences are not directly dependent on the high-level implementation of the basic operations, but on the number of operations that need to be performed and whether they are performed in hard- or software. Security is an important issue, therefore hardware is preferred. Decreasing the number of inversions, gates spent to accelerate the inversion can be used to increase the performance of a multiplication. This is necessary because the number of multiplications is increased using projective coordinates (Appendix N).

With this configuration it is possible to adapt the calculation sequence of a scalar multiplication in such a way that the number of operations can be reduced enormously. This is the Montgomery scalar multiplication and is discussed in detail in Paragraph 5.4 [21].

5.3.3 Normal based with affine coordinates

Using a normal basis with affine coordinates it is possible to calculate a multiplication with a Massey-Omura multiplier [17] or a Massey-Omura-based multiplier [20]. A disadvantage of the Massey-Omura multiplier is that each reduction polynomial needs a different configuration of the multiplier. With multiplexing it is possible to one solution for multiple polynomials, but this solution uses many gates. Another multiplier consists of two parts. A systolic array, for the formation of the coefficient
matrix and a double-rectangular processor. These two parts are one column in the bit-serial multiplier. The number of columns that will be used is equal to the number of bits from the reduction polynomial. This results in a lot of gates and cycles. The normal based configuration with affine coordinates is not advised due to the enormous number of gates that is required to implement a full multiplication.

5.3.4 Normal based with projective coordinates

Another option is a normal based implementation with projective coordinates. The multiplier can be designed with two Massey-Omura matrixes [17]: a parallel input - serial output architecture and a serial input - parallel output architecture. It can be designed with a Bit Serial Systolic Multiplier [20]. The disadvantage of the two multipliers is that they are both not scalable. For a value of m-bits a width and height of m are needed. This requires a relative large architecture for large m. In comparison to the other implementations the number of gates is not enormous. Although when multiple reduction polynomials will be implemented, multiple architectures have to be designed. Implementing a normal based projective coordinates solution for multiple reduction polynomials is therefore not recommended. Whether the scalability is minimal or the number of gates is large.

5.4 Montgomery scalar multiplication (using projective coordinates)

The scalar multiplication explained in Paragraph 2.3.4 could be optimized using a Montgomery scalar multiplication [21]. Point-doublings and -addings consists of less basis operations when Montgomery is used. The execution sequence for doubling and adding is shown in Appendix O. Algorithm 5-1 shows the pseudo-code for a Montgomery scalar multiplication. It is clear that the number of point-addings is equal to the number of bits instead of the Hemming weight of the scalar.

Algorithm 5-1, Montgomery scalar multiplication with example

```
//CALCULATE Q = s · P
P => (Xp, Yp, Zp)
s => (Sm, ..., Sz, S0), with S_m = 1
Q => (Xq, Yq, Zq)
R => (Xr, Yr, Zr)

Q = P;
R = 2P
for (int i = m-1; i >= 0; --i)
{
    if (Si == 1) {
        Q = Q + R;
        R = 2R;
    } else {
        R = Q + R;
        Q = 2Q;
    }
}
```

This increases the number of point operations enormously (in average cases). The number of basic operations per point-operation is decreased. This reduces the total number of basic operations per
scalar multiplication. A comparison of the number of cycles per configuration is made in Appendix P. The resistance against power and timing attacks is discussed in the next paragraph.

5.5 Countermeasures against power and timing attacks

Countermeasures against power and timing attacks that overtake the key or message are investigated. In the elliptic curve implementation a point-adding and a point-doubling should consist of the same operations. The power usage and the execution sequences should not depend on the number of ones and zeros in the polynomials. Execution of a point-adding means that a one is detected in the scalar. Therefore it is desired that a point-adding can not be distinguished from a doubling. The sequences for point-adding and point-doubling for both coordinates systems are summarized in Appendix O. In Appendix P the adapted execution sequences for affine and projective coordinates are shown together with the execution sequence using projective Montgomery. Fake-states are inserted to make both point-operations similar and not identifiable by power and timing analysis. In Paragraph 5.7 the differences in performance with and without these security steps are discussed in detail. If Montgomery calculations are performed no fake-states need to be added, because the point-operations are independent of the bits from the scalar. Point-adding and point-doubling are performed per bit from the scalar.

5.6 Comparison of different configurations

To compare the different implementations estimations are made for the number of gates and the number of cycles necessary for executing a scalar multiplication. For all implementations the reduction polynomial has a maximum size of 233-bits. To make a useful comparison, the number of gates and corresponding performances are estimated for the same number of bits. The polynomial base implementations are scalable for multiple polynomials. Decreasing the number of gates from the multiplier, increases the number of cycles for a multiplication. In the polynomial based configuration is chosen for a combined multiplier and adder. This reduces the number of gates enormously and barely increases the number of cycles. The number of gates for normal basis configuration is fixed for a certain reduction polynomial. When another polynomial is used a similar multiplier has to be designed. The advantages and disadvantages of the four configurations are shown in Table 5-1 and Table 5-2. In the second table three different implementations are compared, these are affine, projective and projective Montgomery implementations.

<table>
<thead>
<tr>
<th>Polynomial</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Scalable multiplier for all polynomials, the area and performance are direct proportional</td>
<td>- Not scalable multiplier</td>
</tr>
<tr>
<td>- Squaring and multiplication are performed similar and fully scalable to the size of the multiplier</td>
<td>- Squaring is simple and takes just one cycle, multiplication is hard and only possible over the full length of the reduction polynomial</td>
</tr>
</tbody>
</table>

Table 5-1, Comparison of different bases

In Appendix Q the results of the estimation for gates and cycles (area and performance) are summarized. The architectures belonging to the configurations are discussed in the next chapter. The number of operations is derived from Appendix O. The gates are NAND-equivalents in TSMC 0.18μ technology. The sizes are the primitive elements and derived from the SAGE™ Standard Cell Library [12].

Copyright © SafeNet. All rights reserved.
### Table 5-2, Comparison of different coordinates systems

<table>
<thead>
<tr>
<th>Affine</th>
<th>Projective</th>
</tr>
</thead>
<tbody>
<tr>
<td>no conversion needed</td>
<td>conversion needed</td>
</tr>
<tr>
<td>one inversion per adding / doubling, this results in $(1 + W_H(\text{scalar})) \times #\text{bits of scalar}$ inversions for a scalar multiplication</td>
<td>a lot of extra multiplications and squarings, but no inversions per adding and doubling, 2 inversion after a scalar multiplication</td>
</tr>
<tr>
<td>number of operations for a point-adding / point-doubling is smaller</td>
<td>more operations needed for a point-adding / point-doubling</td>
</tr>
</tbody>
</table>

**W_H(\text{scalar}) point-addings:**
- 9 additions, 1 cycle
- 2 multiplications, m cycles
- 1 squaring, m cycles
- **1 inversion, 6 x m cycles**

**m point-doublings:**
- 6 additions, 1 cycle
- 2 multiplications, m cycles
- 1 squaring, m cycles
- **1 inversion, 6 x m cycles**

**W_H(\text{scalar}) point-doublings:**
- 7 additions, 1 cycle
- 15 multiplication, m cycles
- 4 squaring, m cycles

**m point-doublings:**
- 4 additions, 1 cycle
- 5 multiplications, m cycles
- 5 squarings, m cycles

**Montgomery**

**W_H(\text{scalar}) point-addings:**
- 3 additions, 1 cycle
- 2 multiplications, m cycles
- 4 squarings, m cycles

**m point-doublings:**
- 1 addition, 1 cycle
- 4 multiplications, m cycles
- 1 squaring, m cycles

Comparison of the different implementations immediately scores out the normal based implementations because they are not scalable for multiple reduction polynomials. When the core has to handle more than one polynomial the number of gates increases enormously. The projective polynomial implementation has a worse performance but when area has to be saved it is possible this configuration becomes interesting due to the relatively small number of inversions. The projective Montgomery implementation has better performance and a similar gate count. Therefore this configuration has the best figures. The remaining solution, the polynomial basis, affine coordinates implementation has a good performance in comparison with the other implementations. Another advantage of polynomial based configurations is the scalability of the multiplier. In the next paragraph all three polynomial configurations will be discussed in detail. In the next chapter different architectures are investigated, the designs with the best performance/gate ratio will be implemented.

### 5.7 Detailed comparison polynomial based ECC-implementations

The comparison made in the previous paragraph results in a preference for a polynomial based implementation. The choice of coordinate system is harder. The necessary area for the operations is similar, but the number of basic operations is different. Inversion is the most important operation, if this operation is completely implemented, with a maximal performance, the situation from Appendix Q is reached. When gates need to be saved it is possible to reduce the inversion control. This
decreases the performance of the affine implementation enormously but has hardly consequences for the projective implementations. In the next table the differences for multiple inversion controllers are shown. The last column shows the performance when wait-states and dummy-loads are added to make power and timing attacks senseless.

<table>
<thead>
<tr>
<th>coordinates</th>
<th>#bits</th>
<th>#gates for inv.</th>
<th>#gates</th>
<th>#cycles</th>
<th>#cycles with waits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projective</td>
<td>160</td>
<td>SW / CPU - 0°</td>
<td>24800</td>
<td>435280</td>
<td>643920</td>
</tr>
<tr>
<td></td>
<td></td>
<td>800</td>
<td>25800</td>
<td>419920</td>
<td>628560</td>
</tr>
<tr>
<td></td>
<td></td>
<td>400</td>
<td>25500</td>
<td>272452</td>
<td>272452</td>
</tr>
<tr>
<td></td>
<td></td>
<td>400</td>
<td>25300</td>
<td>467360</td>
<td>472800</td>
</tr>
<tr>
<td></td>
<td></td>
<td>800</td>
<td>25800</td>
<td>313760</td>
<td>315600</td>
</tr>
<tr>
<td>RSA***</td>
<td>1024</td>
<td>N/A</td>
<td>45000</td>
<td>1492447</td>
<td>N/A</td>
</tr>
<tr>
<td>Projective</td>
<td>192</td>
<td>SW / CPU - 0°</td>
<td>28000</td>
<td>626496</td>
<td>926784</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>29100</td>
<td>603456</td>
<td>903744</td>
</tr>
<tr>
<td></td>
<td></td>
<td>500</td>
<td>28800</td>
<td>393612</td>
<td>393612</td>
</tr>
<tr>
<td></td>
<td></td>
<td>500</td>
<td>28600</td>
<td>672576</td>
<td>680256</td>
</tr>
<tr>
<td>RSA***</td>
<td>2048</td>
<td>N/A</td>
<td>45000</td>
<td>1136528</td>
<td>N/A</td>
</tr>
<tr>
<td>Projective</td>
<td>224</td>
<td>SW / CPU - 0°</td>
<td>30600</td>
<td>852432</td>
<td>1261008</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1150</td>
<td>31900</td>
<td>820176</td>
<td>1228752</td>
</tr>
<tr>
<td></td>
<td></td>
<td>575</td>
<td>31600</td>
<td>534476</td>
<td>534476</td>
</tr>
<tr>
<td></td>
<td></td>
<td>575</td>
<td>31400</td>
<td>915040</td>
<td>925344</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1150</td>
<td>31900</td>
<td>613984</td>
<td>617232</td>
</tr>
<tr>
<td>Projective</td>
<td>288</td>
<td>SW / CPU - 0°</td>
<td>36000</td>
<td>1408464</td>
<td>2083536</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1500</td>
<td>37800</td>
<td>1353168</td>
<td>2028240</td>
</tr>
<tr>
<td></td>
<td></td>
<td>750</td>
<td>37300</td>
<td>880716</td>
<td>880716</td>
</tr>
<tr>
<td></td>
<td></td>
<td>750</td>
<td>37000</td>
<td>1511712</td>
<td>1528416</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1500</td>
<td>37800</td>
<td>1014048</td>
<td>1019088</td>
</tr>
</tbody>
</table>

Table 5-3, Performance figures polynomial bases elliptic curve implementations

* Note: Using this configuration the inversion is performed external. The software triggers the CPU to execute the inversion. This takes about m/32 times more cycles than an inversion performed by the coprocessor

**Note: Fake states (data-load and -stores, additions and multiplications) are added to make power and timing attacks senseless

***Note: The security level of 1024 bits RSA is equal to 160 bits in elliptic curve cryptography, 2048 bits in RSA are equal to 210 bits in elliptic curve [15]

The table above gives the performance figures for a multiplier with depth one. Increasing the depth of the multiplier matrix the performance of the three projective (including Montgomery) implementations improves more in comparison with the two affine implementations.

5.8 Other implementation issues

From the figures of Table Q-5 (Appendix Q) it can be noticed immediately that the number of large registers is doubled in comparison with the PIP. This is possible due to the relatively small word-sizes of elliptic curves (Table 2-1). Therefore the number of memory accesses is reduced to a minimum. The power dissipation is reduced and the execution speed is increased. Despite of the relatively large number of gates of a register the profit of using a register instead of accessing the RAM-memory is
large. Loading and storing the values over a 32-bit bus causes a lot of overhead in comparison to the time that is necessary for an operation. For example a 210-bit addition takes one cycle. Loading and storing the 233-bit values from and to the RAM takes \( 3 \cdot \left\lceil \frac{233}{32} \right\rceil = 24 \) cycles per addition.

The longest path has to be detected, this path must have a smaller delay than \( \frac{1}{\text{frequency}} \). In our design the frequency will be about 20MHz. Therefore the delay through the longest path must be smaller than 50ns. In this implementation the longest path will be in the multiplier architecture where the depth of the matrix and the buffering of the signals cause a large delay. In the next chapter the delay through the longest paths will be discussed in detail.
6. Design of binary field elliptic curve solution for a smartcard core

6.1 Introduction

Two configurations are designed, affine coordinates and projective coordinates using Montgomery, both with polynomial basis. Every design step has to be checked whether the implementation could be easily adapted to a faster or smaller design. This is necessary because it is preferred that the design is scalable for future implementations. Both configurations should be easily exchangeable in the design. The operations remain the same only the control module has to be replaced and the register module has to be adapted. For another basis a totally different design has to be developed since the performed operations are completely different.

In the first paragraph the high-level structure of the cryptography core is developed. The basic blocks of the core are contrived and the functionality of the module is explained. In the second paragraph the exact operations in a polynomial based environment will be explained. In the following paragraph the high-level modules are divided into sub-modules down to basic elements. XOR- and AND-ports, multiplexers and registers are examples of the basic elements. In the next paragraph the finite state machines (FSM) of the control modules are developed. Finally the performance, synthesis and the scalability are discussed.

6.2 High-level architecture

One of the advantages of the polynomial basis structures is the scalability of the architecture and the adaptability for multiple polynomials. Addition and multiplication can be combined in one multiplier-component. This is a matrix architecture for polynomial multiplications. A part of the multiplier can be used for addition and subtraction. A squaring can be performed with the same multiplier. Inversion requires some extra control and registers, the calculations that have to be executed can be performed with the adder/multiplier structure. All the operations will be discussed in detail in the next paragraph. Except hardware that performs the operations, control logic is necessary that triggers the right registers and the execution modules.

Figure 6-1, High-level architecture of the elliptic curve cryptography core
The number of registers that is desired to perform the operations is \((x+1)\)-times \(m\)-bit and \(y\) address-pointers. The maximum number of bits of a polynomial is \(m\), \(x\) is the number of temporary \(m\)-bit values and \(y\) is the number of values that is stored in the RAM. Finally an interface has to be developed that controls the communication with the RISC-processor and the memory. The main architecture that arises is shown in Figure 6-1. The control module contains three sub-modules. One module is necessary for the control of the complete scalar multiplication. Two others control the two different point-operations (adding and doubling). These two point-operations are the main operations of the elliptic scalar multiplication (Paragraph 2.3.4). Implementing the two different configurations the basic- and execute-module are similar. The control module is replaced completely and in the register module two extra register are added.

6.3 Basic polynomial based operations in binary field

The number of basic operations that are necessary executing one adding or doubling in case of affine coordinates or projective coordinates with Montgomery is already discussed. The sequences for these operations can be found in Appendix O. The basic operations with a polynomial basis are addition, subtraction, multiplication, squaring and inversion.

6.3.1 Polynomial addition and subtraction

Polynomial additions within the binary field are similar to subtractions. Binary addition or subtraction modulo 2 is equal. Further more, addition or subtraction of two polynomials always results in a new polynomial with a combination of factors that already exists in the two operands. For example:

\[((x^2+1) + (x^4+x^2+x+1)) \mod 2\] results in \((x^5+x)\). More in general, addition and subtraction of polynomials modulo 2 can be performed by bit-wise XOR of both operands. The pseudo-code for addition is shown in Algorithm 6-1. The modular polynomial is not used because the result is always smaller than the reduction polynomial when both operands are smaller than the reduction polynomial.

Algorithm 6-1, Polynomial addition in a binary field

\[
//CALCULATE: Xr = Xp + Xq \mod Xf
\]

\[
Xp \equiv U \rightarrow (U_{m-1}, ..., U_1, U_0), Xq \equiv V \rightarrow (V_{m-1}, ..., V_1, V_0)
\]

\[
Xf \equiv F \rightarrow (F_{m}, F_{m-1}, ..., F_1, F_0), Xr \equiv C \rightarrow (C_{m-1}, ..., C_1, C_0)
\]

\[
for \ (i = 0; \ i < m; \ ++i)
\]

\[
Ci = Ui \oplus Vi;
\]

6.3.2 Polynomial multiplication and squaring

The polynomial modular multiplication is more complex than the addition. The reduction polynomial has to be taken into account because the product could have components up to \(x^{2m-2}\). Therefore the multiplication is performed by shifting and adding the polynomials through a matrix structure. In this structure the reduction polynomial is subtracted every time the result becomes larger than \(m\) bits. The pseudo-code for the multiplication is shown in Algorithm 6-2. This algorithm can be performed in one cycle when a matrix is designed with a size of \((m-1) \times (m-1)\). Practically this is not the best solution because the number of gates is large and a data path arises that contains a lot of elements. The size of the matrix is dependent of the maximum number of gates and the desired performance. The number of cycles for a multiplication is directly related to the size of the matrix. The multiplication is a sequence of XOR- and AND-operations, using only the XOR-operators an addition can be performed by the
multiplier. The result of the addition is available at the outputs of the matrix at the first row. The architecture of the multiplier is discussed in detail in the next paragraph. The multiplier also performs a squaring operation. Squaring is performed similar to a multiplication of two different polynomials and has therefore the same performance.

Algorithm 6-2, Polynomial multiplication in a binary field

```
//CALCULATE: Xr = Xp · Xq mod Xf
Xp = U → (Um-1, ..., U1, U0), Xq = V → (Vm-1, ..., V1, V0), Xf = F → (Fm, Fm-1, ..., Fl, F0), Xr = C → (Cm-1; ..., Cl; C0)
#define C-1 0
for (int i = 0; i < m; ++i)
    Ci = 0;
for (int i = m-1; i >= 0; ++i)
{
    bool temp = Cm-1;
    for (int j = m-1; j >= 0; ++j)
        Cj = ((Vj ⊕ Uj) ⊕ (Fj ⊕ temp)) ⊕ Cj-1;
}
```

6.3.3 Polynomial inversion

With polynomial basis an inversion can be performed by shifting and adding the operands with the reduction polynomial. The inversion continuously compares the adapted values of the operand and the reduction polynomial. Meanwhile the operands and the polynomial are shifted right and added with temporary values.

Algorithm 6-3, Polynomial inversion in a binary field (based on the euclidean algorithm)

```
//CALCULATE: Xr = Xp/Xq mod Xf
Xp = W → (Wm-1, ..., W1, W0), Xq = Y → (Ym-1, ..., Y1, Y0)
Xf = N = X → (Xm, Xm-1, ..., X1, X0), Xr = Z → (Zm-1; ..., Z1; Z0)
Z = 0;
while (W != 0)
{
    while (W0 == 0 || X0 == 0)
    {
        if(W0 == 0)
        {
            W = W >> 1;
            if(Y0 == 1)
                Y = Y + N;
            Y = Y >> 1;
        }
        else
        {
            X = W + X;
            Z = Y + Z;
        }
    }
    if(W == X)
    {
        W = W + X;
        Y = Y + Z;
    }
    else
    {
        X = W + X;
        Z = Y + Z;
    }
}
```
Executing the inversion, two additions are performed in parallel. This is necessary to update the reduction polynomial. The pseudo-code for the inversion is shown in Algorithm 6-3. The Euclidean algorithm for polynomials [2] is the basis for the inversion. Using the inversion algorithm, parallel with the inversion a polynomial could be multiplied with the inverted polynomial. This reduces the number of multiplications. When this is not desired make $X_p$ one (Algorithm 6-3). If it is possible to execute two additions parallel the performance of a single inversion doubles.

### 6.4 Low-level architecture

The first step towards a detailed hardware implementation is to make a specified architecture from the blocks of Figure 6-1. The interface module can be copied from the PIP hardware implementation, the other modules have to be completely designed. The number of registers that is necessary depends on the configuration (Appendix Q). For all configurations the number of m-bit registers that are relatively large is constrained to a minimum. Executing an inversion four operands need to be available continuously: $W$, $X$, $Y$ and $Z$ (Algorithm 6-3).

The basic operations are similar for the configurations that are investigated in detail. Therefore the basic- and execution-module can be implemented in detail. These modules perform the basic operations (addition, multiplication and inversion).
The control-module is completely dependent on the configuration. The interface towards the other modules is similar, but the internal architecture has to be designed per configuration. In Figure 6-2 the basic-, execution- and register-module are shown in detail. The gray blocks will be modules (entities, VHDL) in the final design. The white blocks inside the gray blocks will be different processes within the modules.

The choice of the multiplier is very important. Dependent on the desired performance a multiplier could be chosen. For polynomial based implementations different multipliers are compared. The most important are: a multiplier-and-adder [17], a shift-and-add multiplier [17] and a matrix multiplier [18]. The matrix multiplier is a systolic version of the shift-and-add multiplier. To save gates, the multiplier can also be used to perform an addition. An example of the multiplier architecture is shown in Figure 6-3. In this figure the width of the matrix is m, the depth of the matrix is two. This means one operand is used completely, the other operand per two bits. The number of passes in this architecture is $m^2/\text{(width·depth)}$. In this case there are $m/2$ passes. The MUX’s are necessary to perform a normal addition and an addition followed by a shift-back action. This operation is necessary when an inversion is performed. The bold input signals will be discussed later.

![Figure 6-3, Architecture of the matrix shift-and-add multiplier (depth of matrix is two)](image)

To manage all the different operations within the multiplier, the different state machines control multiple signals. The multiplexing towards the multiplier is complex, because all registers could provide data for the multiplier inputs.

### 6.5 Area and performance estimations for the implemented ECC design

In this paragraph a detailed estimation for the number of cycles and gates per configuration is made. The basic module and execution module are independent of the configuration. The most important requirement is the number of gates, therefore the depth of the matrix is chosen one. The width of the matrix is m, if the width is decreased the control logic increases enormously. These depth and width are used in both elliptic curve implementations discussed in this thesis.

The chosen value for m is 233, this results in the following area and performance estimations. The estimated number of cycles for affine coordinates is about 655 thousand cycles for a complete scalar multiplication. The number of gates is estimated on 32k5 for a 233-bit implementation. In the
projective Montgomery case performance and area are respectively estimated on 556 thousand cycles and 32k gates. These estimations are calculated similar to the estimations made in Appendix Q, only for \( m \) is 233 bit.

### 6.6 VHDL implementation of the elliptic curve design

#### 6.6.1 Similarity with the PIP2025 hardware

The elliptic curve implementation and the PIP have the same interface towards the memory and the RISC-processor. Therefore it is useful to use the interface module from the PIP design and adapt it towards this implementation. Furthermore no PIP hardware can be used in the elliptic curve design. The timing diagrams of the PIP are useful for the external communication with the memory and the RISC-processor.

#### 6.6.2 Control signals between the high-level modules

The control module generates signals that control the basic operations executed by the multiplier and controlled by the basic module. To prevent timing problems signals are stored in registers after passing the basic module. One cycle later the execution module is controlled by the stored signals. This shortens the critical path through the design. The operation that uses the temporary results to determine the next step is the inversion. The control module for this operation should take into account that the results of the operations are not immediately available. Adding an extra state, in which the calculated result is compared with another value, solves this timing problem. For all the other operations no problems occur because the temporary results are not used to determine the next calculation.

Signals from the interface have to trigger the registers when data needs to be stored. In case of a CDP-instruction the control or basic module has to be triggered. From the interface to the registers a three-bit signal indicates the register. Another three bits indicate the location in the specific register. Executing a CDP-instruction a 7-bit signal triggers the control module. The interface decodes the instructions. To prevent a decode block in the control module, the interface passes the decoded instruction by separate trigger signals.

All the signals towards the execution module are buffered in the basic module, independent whether they are generated in the control module or the basic module itself. These signals select the input data from the registers and configure the multiplier to execute the desired operation.

The control and basic module have some internal signals that control the state machine. In the control module these signals trigger the point operations and determine which state is entered. In case of an inversion the basic module generates six control signals for determination of the next state. Finally the basic module has a two-bit signal that selects a result register.

The control signals are summarized in Appendix S. In the appendix the names of the signals and its values are described.

#### 6.6.3 Detailed timing diagrams for all possible situations

In the developed timing diagrams the signals and register values are visualized. For all basic operations a timing diagram is developed. There is one major difference between an addition and the other operations. The control module (state diagram of point-operations) controls an addition, the basic module controls the other operations (inversion and multiplication). These operations take two
extra cycles in comparison with the optimal situation. One cycle to trigger the state diagram in the basic module and one cycle to indicate the operation is finished.

In Appendix T the timing diagrams for the basic operations are shown. In the timing diagrams the signals correspond to the input signals of the multiplier. The control- and basic-module generate the signals. The signals are stored in a register and one-clock cycle later they trigger the multiplier. The external timing of the interface is similar to the PIP, it can be found in the ARM architectural reference manual [5].

6.6.4 Implementation of the (sub-)modules

In this paragraph the implementations will be discussed down to the VHDL-code. Dependent on the desired functionality, the VHDL-code represents finite state machines, combinatorial logic or sequential logic. The control logic will be designed as FSM's, the execution part is designed as combinatorial logic and the registers in all the modules are of course part of sequential logic.

6.6.4.1 The control module for both configurations

The two most interesting configurations (affine and projective Montgomery) have complete different execution sequences (Appendix O). Therefore two different control modules have to be designed. The most complex sub-module is for both configurations the scalar multiplication. The point-operations are mostly straightforward execution sequences.

Due to the complexity of the scalar multiplication is has to be designed by a FSM. Although for the point-operations different hardware implementations can be developed. These are a large but straightforward FSM and a solution with micro-code. When the point operations are implemented with micro-code a local ROM contains the execution sequence. Just one register is needed to indicate which line has to be executed. This line contains the values of the control signals to the other modules or a possible jump action. The other solution, a FSM, also needs a register, in this case to indicate which state is executed. Dependent on the input signals and the value of the register the control signals are configured with logic. To determine which implementation is the best, the number of gates necessary to implement the ROM is compared with an estimation of gates for the logic necessary to generate the control signals.

The ROM has 24 lines with 21 bit data that results in a 504-bit ROM. The smallest possible ROM is 256 byte and has a size of 2k gates. Except for a ROM, this solution needs a power ring to feed the ROM, this is coupled to two power-paths. Finally an interface is necessary to access and test the ROM. The overhead of the ROM is about 1k gates. The complete solution is 3k gates per point operation.

In case of a FSM the gates for the control logic is at most 5×AND and 2.5×NOT per state per signal. That results in about 2650 gates (NAND-equivalent) worst case. The FSM is simple and can be optimized. This halves the number of gates and result in an implementation with 1k3 gates.

Comparing both solutions the FSM is the best option. Furthermore implementing a ROM, license-rights have to be paid. These two arguments result in the implementation of the FSM solution. The FSM's of the affine and projective Montgomery scalar multiplication are shown in Appendix U1 and U2. The state diagrams for adding and doubling can be directly derived from the execution sequences in Appendix O. There are two extra sub-modules implemented to control the signals to and from the doubling and adding state machines. In these sub-modules we divided the different signals into several processes to decrease the simulation time.
6.6.4.2 The basic module

The basic module contains two FSM's and logic to determine the input signals for the inversion state machine. The other state chart involves the multiplication, the extra logic beside the state machine is a register that indicates the passes through the multiplier. For the inversion six different signals are generated. These signals determine the next state. In Appendix S these signals are explained.

All the signals from the control to the execution module pass the basic module and are selected differently per operation. In case of an inversion all the signals are generated by the inversion FSM and thus overrule the control signals from the control module. In all the other cases only the counter value, SBA and the register reset are generated locally. The other signals are generated by the control module and passed by the basic module. All the output signals generated internal or by the control module, are buffered in a register in the basic module. This is done to decrease the longest path. In this way two different paths are created, a control-path through the control and basic module and a data-path through the execution module. The longest control path starts in the control module with the generation of MAI, it triggers the FSM's in the basic module and determines the signals to the multiplier. The FSM's of the basic module are shown in Appendix U3.

6.6.4.3 The execution module

In the execution module data is selected from the registers with the generated control signals. With this data the calculation is performed and the result is offered to the registers. The calculation mainly involves AND- and XOR-operations and multiplexing of the preliminary result. The depth of the matrix structure is one. This configuration is chosen because the area is more important than the performance for this implementation.

![Figure 6-4](image)

Figure 6-4, Final architecture of the multiplier (width is M, depth is 1)

Around this module a multiplexing module is located that multiplexes the data to the desired inputs. All the inputs of the complete multiplication module are directly from a register. This counts for signals as well as data. Therefore the longest path is the path of the data from the registers through the multiplier and back to the registers.

6.6.4.4 The register module for both configurations

The register module consists mainly of registers and multiplexers. Except that, logic is necessary to trigger the right multiplexers dependent on the input signals. The register consists of two parts. One
part contains the temporary registers and the reduction polynomial. The width of these registers is \( m \). The other part contains the registers of maximal 32-bit. These are pointers, constant numbers and 32-bit of the scalar. Dependent on the configuration, seven or nine 14-bit pointers are necessary. The 14-bit pointer points to a base RAM-addresses. The remaining registers are two 8-bit registers and a 3-bit register to hold three constant values. The registers are accessible by the interface to be re-written. The execution module can only access the four temporary \( m \)-bit registers. When one of this registers is read, first one of the \( m \)-bit values is selected and passed trough the execution module. After selecting the \( m \)-bit value, a 32-bit value is selected and stored in a register. The architecture necessary to trigger the right registers and the data-paths in the register module are shown in Appendix V. The register module differs per configuration. With affine coordinates the s-word register is loaded per 16-bit. In the projective Montgomery case the register is loaded per complete 32-bit word. The main difference is the need for two extra data-pointers in the projective Montgomery case. From three points the base-addresses of an \( x \)- and \( z \)-coordinate have to be stored. In the affine case just four base-addresses have to be stored, for two points an \( x \)- and \( y \)-coordinate.

6.6.4.5 The interface module

The part of the interface that controls the data- and addressbus is copied from the PIP. After that it is modified for all the CDP instructions. To load and store data during calculations, a process in the interface that controls the loading from or storing into the memory is added. The addresses of the coordinates will be generated with the register-module.

A more detailed description of modules, finite state machine and implementation issues can be read in the system specification [22].

6.7 Testing, simulation and synthesis

6.7.1 Testing of the design

The implemented design is divided into five sub-modules on the highest level. The high-level sub-modules (interface, control, basic, registers and execution) are tested separately. Most of these modules contain sub-modules itself. The modules are combined when all composed tests are performed successfully. In theory the developed tests should cover 100% of the input combinations, in all possible states of the module. In practice this is hardly possible, although the optimal situation has to be approached. The tests are composed in such a way all the states within the modules are accessed and all transitions are taken. Finally the modules are combined and complete test vectors are composed. These vectors test the complete functionality of the elliptic curve cryptographic core. To make sure the tests cover most of the situations a navigator tool is used. This tool mathematically determines which percentage of all input-state-combinations is tested. The test-benches are developed with standard SafeNet test-bench modules. These modules make it possible to read an input command-file with test vectors and clock commands. The generated outputs can be checked by separate test-modules. The results of the simulation are logged and stored in a log-file. For the two different configurations just one reusable test-bench is developed. Dependent on the configuration a command-file has to be loaded. The ports from the test-modules that read outputs are set on high-impedance (HIZ). The clock is started and the inputs are set on their initial value. From that moment input vectors are generated and output values are checked. Finally the clock is stopped and the simulation ends. In the log-file the
inputs, clock values and check-results are shown. Per check the read value is compared with the expected value and indicated whether the check result is o.k. or the check has failed. Naturally no checks may fail to pass the complete test.

6.7.2 Simulation and syntheses results

The implemented designs are the 233-bit versions of affine and projective Montgomery coordinates within the binary field. The performance results of the simulations are in a reasonable range of the expected values (Figure 6-5). The number of cycles is larger due to overhead of data-loads and -stores that were measured during calculations. Implementing the design it seemed that many gates could be saved when the loads and stores are not always performed during a calculation. The multiplexing in the register module is reduced because hardware can be shared. The total number of cycles is increased, but the gate count is much less. Besides the reduction of gates in the registers, the imported interface module is halved in comparison with the initial estimations. In total the number of gates is reduced with 10%, against an increment of the number of cycles with 5%. The developed implementation is preferred over the initial design because the area usage is the most important issue. For the final design and the final load and store sequences is referred to the system specification [22].

<table>
<thead>
<tr>
<th>configuration</th>
<th>performance (cycles)</th>
<th>gate count (NAND eq.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>expected</td>
<td>simulated*</td>
</tr>
<tr>
<td>affine</td>
<td>655.000</td>
<td>674.088</td>
</tr>
<tr>
<td>projective Montgomery</td>
<td>556.000</td>
<td>591.200</td>
</tr>
</tbody>
</table>

Figure 6-5, Comparison of expected and actual performance figures and gate count

* Note: The number of cycles for an inversion differs dependent on the bit-sequence. Therefore different test-vectors could give different results. Theoretically the result differs maximal 25% in the affine case and 0.1% in the projective Montgomery case. In the affine case the number of inversions is large, also the operands of the inversions change continuously. Therefore the mathematically spreading of the inversions within one scalar multiplication is large. This concludes that the simulation result of a complete scalar multiplication approaches the average value. The scalar has an average number of ones (in the affine case the performance could vary 25% up and down, dependent on the number of ones).
Conclusions

Conclusions towards the software model of the PIP2025

Development of a software model gives the opportunity to describe and implement the complete functionality of a new hardware design. Due to the cycle accuracy of the model, it is possible to measure exact performance figures. The model is scalable and configurable, therefore it is suitable to measure multiple performance figures and choose the best design. The followed strategy of developing detailed class models with a detailed class description turned out to be a practical method to develop and maintain the software model. The system specification and relative algorithms are studied in detail during the development of the class models.

Besides performance measurement, the developed software model turned out to be very useful for comprehension of the product and creation of VHDL command-files. Next to the model for internal use, customers can use a model with less-detailed information. Customers can simulate the same instructions with the model as with the actual design. The model gives a visual impression of the system and is useful understanding the product.

Furthermore the software model is able to generate VHDL command-files complete with input-vectors and output-checks. These can be used simulating a part of the coprocessor VHDL-design. Finally the model can be used to verify results of complete simulations and do pre-emptive simulations with border-case-simulations. The model is useful to develop test-vectors for the actual design.

The advantages summarized above make it very useful to develop a software model of a new design.

Conclusions towards the binary field elliptic curve design

Several possible implementations for an elliptic curve cryptography system has been investigated. This study resulted in two algorithms that could be implemented efficiently:

- binary field, polynomial basis with affine coordinates;
- binary field, polynomial basis with projective coordinates using Montgomery.

A detailed investigation for the possible configurations results in better performance figures and small area usage in binary field elliptic curve implementations. Besides these two advantages the scalability is the third major advantage designing polynomial based configurations. The other configurations were only competitive on a segment of these issues. The possibility to implement two designs, affine and projective, enlarges the application options to use these implementations. Therefore is chosen to implement exchangeable modules that make it possible to switch between both implementations. Due to the small key-sizes of elliptic curve solutions, the designed system uses less memory and dissipates less power in comparison with an RSA solution.

General conclusions towards the used strategies of the ECC design

Having the RSA implementation in mind, the elliptic curve implementation has been developed. Low-cost implementation issues of exchangeable modules where already tackled. The choice for the two configurations is made according to detailed area/performance estimations. To give notice to the area issue we were focussed to keep the hardware implementation as small as possible. Therefore registers
are combined and extra effort is taken to combine operations into the same hardware module. This resulted in a small but scalable execution module that performs all binary field polynomial based operations.

At first the elliptic curve implementation has been tested per module. The advantage of this test method is that the functionality of the modules is completely tested before the modules are combined into the final design. The complete test is aimed on the timing between the modules. With the timing is meant whether signals toggle at the right time and the availability of data. The test results were similar to the expected performances. Synthesis of the design gave the expected area usage. These conclusions confirm the improvement using elliptic curve cryptography instead of today's cryptographic standards.
Recommendations

Recommendations for further development of the PIP2025 model and future models

With the PIP2025 software model it is currently possible to step through the simulation per coprocessor command with a certain time-delay. Of course, using the development tool, every simulation can be debugged and stopped with breakpoints at any time at any moment. Although a useful expansion of the model can be a debug button in the model itself and/or program instruction that makes it possible to start and stop the simulation at any time and let it executed with a continues adaptable delay. This addition does not improve the actual functionality of the model, but increases the possibility to debug the implementation not only with the trace-file, but also with a parallel running simulation.

Recommendations for extension of the elliptic curve cryptography core

The developed smartcard coprocessor core is designed to be as low-cost as possible, although with acceptable performance. For future implementations and improvements a scalable implementation is developed. The same multiplier could be resized for other designs. The architecture can be used for parallel and/or pipelined multiplier modules, this is very interesting for high-end (low-volume) implementations.

Furthermore the implementation study resulted into a recommendation for binary field polynomial based implementations. Area usage, performance figures and scalability resulted in the recommendation for this configuration. The decision to choose between the affine or projective Montgomery implementation is up to the customer. Implementing both solutions on the same chip is possible, but not recommendable due to the extra area usage of the control module.

Future study is necessary to investigate the implementation of an extra register bank. Three extra m-bit registers reduce the memory access to a minimum. During a CDP-instruction no memory access will be necessary, through which the power dissipation is decreased enormously. This solution is more secure and increases the performance, of course the area usage is increased due to the extra registers.

Dependent on the technology, a register-file for the large registers could be considered to reduce the number of gates. For further reduction, it is possible to remove the registers in the multiplier. The temporary multiplier results have to be stored in one of the other registers. This solution results in extra multiplexing and extra memory accesses, although the total gate count is decreased.

Future cryptographic solutions for smartcard-implementations

The elliptic curve solutions perform much better on all levels in comparison with other currently often used cryptographic algorithms. Figures for gate count, performance and power dissipation give better results than any currently known cryptography system. Therefore elliptic curve cryptography can be seen as the future solution for cryptographic implementations. Which elliptic curve configuration will be processed depends on future customer demands. The choice of the elliptic field depends on customer needs, prime field calculations can be performed on current designs. For binary field implementations the developed elliptic curve design can be used. Within the binary field the preference goes to polynomial based implementations due to the scalability of the multiplier and the relatively simple basis operations. The choice of coordinate system could again be made by the customer and does not influence the gate count and performance enormously.
The project took place at SafeNet B.V. in Vught in cooperation with the research chair Computer, Networks and Design (CND). This chair is part of the Information and Communication System Group (ICS) of the faculty Electrical Engineering of the Technical University of Eindhoven.

We enjoyed the time working on the project and will employ the knowledge we gathered during study, discussions and designing.

Special thanks go to:
Henk Janssen and Hans Eichbaum from the SafeNet IP-group,
Cees Jansen, Geeke Muurling, Lejla Batina and Sheng Bo Xu from the SafeNet Cryptography-group,
George Szanto, product manager of SafeNet,
Mario Stevens and Ad Verschueren from the CND-chair of the University
and
Rian van Gaalen from the ICS-group of the University.

Willem Cuppens
Gijs Willemse
Implementing Cryptographic Solutions in a Modular SmartCard core

Appendices

CONFIDENTIAL
Appendices
Implementing Cryptographic Solutions in a Modular SmartCard core

These appendices belong to the thesis "Implementing Cryptographic Solutions in a Modular SmartCard core — RSA hardware modeling and simulation - Elliptic Curves design and implementation"

ing. W.H.P. Cuppens
G.A.J. Willemse

Graduation project,
Eindhoven, University of Technology
September 2001 – June 2002

Supervisors:
SafeNet, Vught
ing. H.P.F. Janssen
dr. ir. C.J.A. Jansen
G. Szanto ba.

Eindhoven, University of Technology
prof. ir. M.P.J. Stevens
dr. ir. A.C. Verschueren

Copyright © SafeNet. All rights reserved.
# Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of contents</td>
<td>3</td>
</tr>
<tr>
<td>List of Figures, Tables and Algorithms within the appendices</td>
<td>5</td>
</tr>
<tr>
<td>Appendix A. RSA example</td>
<td>6</td>
</tr>
<tr>
<td>Appendix B. Elliptic Curve Example</td>
<td>7</td>
</tr>
<tr>
<td>Appendix C. Montgomery example in RSA</td>
<td>9</td>
</tr>
<tr>
<td>Appendix D. Modular Montgomery Exponentiation example</td>
<td>10</td>
</tr>
<tr>
<td>Appendix E. Coprocessor instructions</td>
<td>11</td>
</tr>
<tr>
<td>E.1 CDP:</td>
<td>11</td>
</tr>
<tr>
<td>E.2 LDC:</td>
<td>11</td>
</tr>
<tr>
<td>E.3 MCR:</td>
<td>11</td>
</tr>
<tr>
<td>E.4 MRC:</td>
<td>12</td>
</tr>
<tr>
<td>E.5 STC:</td>
<td>12</td>
</tr>
<tr>
<td>Appendix F. Coprocessor instructions timetable</td>
<td>13</td>
</tr>
<tr>
<td>Appendix G. Class diagram of the PIP2025 software model</td>
<td>14</td>
</tr>
<tr>
<td>Appendix H. Systolic Array</td>
<td>16</td>
</tr>
<tr>
<td>Appendix I. Architecture PE's</td>
<td>17</td>
</tr>
<tr>
<td>Appendix J. Software model</td>
<td>19</td>
</tr>
<tr>
<td>J.1 Customer</td>
<td>19</td>
</tr>
<tr>
<td>J.2 Internal</td>
<td>21</td>
</tr>
<tr>
<td>J.3 Menu structure of the smartcard simulation</td>
<td>23</td>
</tr>
<tr>
<td>J.4 Program.prg example</td>
<td>24</td>
</tr>
<tr>
<td>J.5 Memory.mem example</td>
<td>24</td>
</tr>
<tr>
<td>J.6 Trace.trc example</td>
<td>25</td>
</tr>
<tr>
<td>Appendix K. Simulation level of internal and customer model</td>
<td>26</td>
</tr>
<tr>
<td>K.1 Simulation Level 0/1 PIP2025 - customer</td>
<td>26</td>
</tr>
<tr>
<td>K.2 Simulation Level 2/3 PIP2025 - internal</td>
<td>26</td>
</tr>
<tr>
<td>Appendix L. Performance figures</td>
<td>28</td>
</tr>
<tr>
<td>Appendix M. Example of a generated VHDL command-file</td>
<td>31</td>
</tr>
<tr>
<td>Appendix N. Class diagrams of elliptic curve representations</td>
<td>32</td>
</tr>
<tr>
<td>N.1 Binary field with affine coordinates</td>
<td>32</td>
</tr>
<tr>
<td>N.2 Binary field with projective coordinates</td>
<td>33</td>
</tr>
<tr>
<td>N.3 Binary field projective coordinates using Montgomery</td>
<td>34</td>
</tr>
<tr>
<td>N.4 Prime field with affine coordinates</td>
<td>35</td>
</tr>
<tr>
<td>N.5 Prime field with projective coordinates</td>
<td>36</td>
</tr>
<tr>
<td>Appendix O. Calculation sequence of different EC representations</td>
<td>37</td>
</tr>
<tr>
<td>O.1 Calculations for BINARY field – AFFINE coordinates</td>
<td>37</td>
</tr>
<tr>
<td>O.2 Calculations for BINARY field – PROJECTIVE coordinates</td>
<td>38</td>
</tr>
<tr>
<td>O.3 Calculations for BINARY field – PROJECTIVE coordinates with MONTGOMERY</td>
<td>40</td>
</tr>
<tr>
<td>O.4 Calculations for PRIME field – AFFINE coordinates</td>
<td>40</td>
</tr>
<tr>
<td>O.5 Calculations for PRIME field – PROJECTIVE coordinates</td>
<td>41</td>
</tr>
<tr>
<td>Appendix P. Sequences to prevent power and timing attacks</td>
<td>44</td>
</tr>
<tr>
<td>P.1 Execution sequence to prevent timing and power attacks</td>
<td>44</td>
</tr>
<tr>
<td>P.2 Formulas for the number of cycles with and without prevention</td>
<td>46</td>
</tr>
<tr>
<td>Appendix Q. Area and performance estimations for different ECC implementations</td>
<td>47</td>
</tr>
<tr>
<td>Appendix R. Elliptic Curve Digital Signature Algorithm</td>
<td>48</td>
</tr>
<tr>
<td>Appendix S. High-level control signals of the ECC-implementation</td>
<td>49</td>
</tr>
</tbody>
</table>
Appendix T. Timing diagrams of the basis operations in polynomial based ECC

T.1 Polynomial Addition

T.2 Polynomial Multiplication

T.3 Polynomial Inversion

Appendix U. State diagrams final implementation

U.1 The scalar FSM in the control module of the affine case

U.2 The scalar FSM of the control module in the projective Montgomery case

U.3 Multiplication FSM of the basic module

U.4 Inversion FSM of the basic module

Appendix V. Global architecture of the register module
List of Figures, Tables and Algorithms within the appendices

List of Figures
Figure B-1, Point adding operation (R = P + Q) ..................................................7
Figure L-1, Performance of PIP2025, average values for a full exponentiation ..........29
Figure L-2, Performance of PIP2025, average values for a full exponentiation ..........29
Figure L-3, Comparison of different configurations with 1024-bit vectors .................30
Figure L-4, Comparison of different configurations with 2048-bit vectors .................30

List of Tables
Table L-1, Performance figures of the PIP2025 software model (two configurations) ....28
Table P-1, Sequence for projective point-addings and point-doublings ................44
Table P-2, Sequence for point-adding and point-doubling (montgomery) ..................45
Table P-3, Sequence for affine point-addings and point-doublings .......................45
Table Q-1, Comparison of different ECC implementations, interesting ones are specified ..47

List of Algorithms
Algorithm O-1, Point-adding with affine coordinates in a binary field ..................37
Algorithm O-2, Point-doubling with affine coordinates in a binary field ..................38
Algorithm O-3, Point-adding with projective coordinates in a binary field using Montgomery .40
Algorithm O-4, Point-doubling with projective coordinates in a binary field using Montgomery .40
Appendix A. RSA example

An example of the RSA algorithm is given below. First the key-generation is described, followed by the en- and decryption algorithms.

Chose two prime numbers P and Q

\[ N = P \times Q \]  

\text{e.g.: } P = 5, Q = 7 \Rightarrow N = 5 \times 7 = 35

Calculate PHI, this is two modules to calculate the private key.

\[ \text{PHI} = (P-1) \times (Q-1) \]  

\text{e.g. } \text{PHI} = 4 \times 6 = 24

PHI can be divided by \([1, 2, 3, 4, 6, 8, 12, 24]\)

A number E can be for example the prime number 5.

The public key is: \(E = 5\) with modules \(N = 35\)

\[ D = 1/E \mod \text{PHI} \]  

Determine D, the private key, using the extended Euclidean algorithm:

chose \(q \) and \(r\) with: \(a = q \times b + r \) and \( r < b \)

\[ \begin{array}{cccccccc}
\text{a} & \text{b} & \text{v} & \text{x} & \text{u} & \text{y} & \text{q} & \text{r} \\
\text{step 1 or (n)} & \text{PHI} & \text{E} & 1 & 1 & 0 & 0 & q_i & r_i \\
\text{step n + 1} & b_i & r_i & y-q \times v & u & x-q \times u & v & q_{i+1} & r_{i+1} \\
\end{array} \]

stop when \( r = 0 \), then collect \( v \) and \( u \)

\[ D = (v > 0 \ ? \ v ; u) \]  

\[ E \times D = 1 \mod \text{PHI} \]  

\text{e.g. } 5 \times D = 1 \mod 24

The encrypted message (cipher-text) of \(M = 2\) becomes:

\[ C = M^E \mod N \]  

\text{e.g. } C = 2^5 \mod 35 = 32

The encrypted message \(C\) will be send from the sender to the receiver

The decrypted message of \(C = 32\), becomes:

\[ M = C^D \mod N \]  

\text{e.g. } M = 32^5 \mod 35 = 2
Appendix B. Elliptic Curve Example

The general high-level calculations of the Elliptic Curves algorithm are described below. Similar to the RSA example, first the key generation is explained, followed by the general en- and decryption algorithms within Elliptic Curves.

Key generation:
1. Select a random integer $d \in [2, n-2]$, with $n$ the field size
2. Calculate $Q = d \cdot P$, with $P$ a point $P(x, y)$ and $Q$ a point $Q(x, y)$
3. The public key is the field $E$, point $P$ and $Q$ and the field size $n$

The multiplication of point 2 is a scalar multiplication. A point has to be multiplied with the scalar $d$. A multiplication is divided into multiple add and double actions of points. When a point has to be added or doubled is already explained in algorithm 2.1 of Paragraph 2.3.4. In Figure B-1 the point-adding is visualized. When a doubling has to be performed a tangent is drawn to the point, further the result is determined in the same way an adding is performed.

![Figure B-1, Point adding operation (R = P + Q)](image-url)
Encryption with hash-function

Have a message $m$ and calculate the encrypted message $c$.

1. Choose $r$ and compute $t = h(m \| r)$, where $\|$ is the concatenation of both operands and $h(x)$ the hashed value of $x$
2. Compute a temporary key-pair $x_V$ and $W$, where $V(x,y) = t \cdot Q$ and $W(x,y) = t \cdot P$
3. Calculate $k = (m||r)$ XOR $x_V$
4. The cipher-text is $c = (k, W)$

Decryption of cipher-text (with hash)

The cipher-text $c$ has to be decrypted into the encrypted message $m$.

1. Compute $x_D$, where $D(x,y) = s \cdot W$, with $W$ is part of the cipher-text and $s$ the secret key
2. Compute $u = k$ XOR $x_D$
3. Check to verify: $W = h(u) \cdot P$, with $P(x,y)$ is a point of the public key
4. The decrypted message are the $m$(length)-MSB's of $u$. The length of $m$ is part of the public key.
Appendix C. Montgomery example in RSA

There are several pre-conditions for a Montgomery multiplication:

Chose a word-width for both operands (a and β) that have to be multiplied: \( \beta \geq 2 \cdot a \).

Calculate \( R = 2^r \) with \( r = \lceil (n+2)/a \rceil \cdot a \geq n+2 \).

Determine \( N' \) for which counts: \( R \cdot R^{-1} \cdot N \cdot N' = 1 \).

In the processing elements the following equations are executed.

\[
\begin{align*}
\{ \text{input } X, Y, N \} \\
T(-1) &= 0 \\
m_1 &= 0 \\
x_k &= 0 \\
\text{for } i = 0 \text{ to } k \\
\quad \text{begin} \\
T(i) &= (T(i-1) + 2^r x_i Y + m_{i-1} N) \div 2^r \\
m_i &= T(i) \cdot N' \mod 2^r \\
\text{end} \\
\{ \text{output } T = T(k) = MMM(X,Y) = X Y \cdot R^{-1} \mod N \}
\end{align*}
\]

An example of \( A \cdot B \mod N \) (in pseudo-code):

\[
\begin{align*}
\text{result}_1 := \text{Mont}(A, R^2 \mod N) \\
\quad \{ = A \cdot (R^2 \mod N) \cdot R^{-1} \mod N = A \cdot R \mod N \} \\
\text{result}_2 := \text{Mont}(B, \text{result}_1) \\
\quad \{ = B \cdot (A \cdot R \mod N) \cdot R^{-1} \mod N = B \cdot A \mod N \} \\
\text{if (result}_2 > N) \\
\quad \{ \text{this is indicated by the status bit} \} \\
\text{result}_2 = \text{result}_2 - N \quad \{ \text{if necessary, calculated in software} \} \\
\text{fi} \\
\text{return result}_2
\end{align*}
\]
Appendix D. Modular Montgomery Exponentiation example

Example 1, an example of an Exponentiation calculation:
Calculation of $x^{1101} \mod N$

\[
A = R \mod N
\]
\[
x' = \text{Mont}(x, R^2 \mod N) = x \cdot R \mod N
\]
\[
A = \text{Mont}(A, A) = R \mod N
\]
\[
e_3 = 1 \rightarrow A = \text{Mont}(A, x') = x^{10} \cdot R \mod N
\]
\[
e_2 = 1 \rightarrow A = \text{Mont}(A, x') = x^{11} \cdot R \mod N
\]
\[
A = \text{Mont}(A, A) = x^{110} \cdot R \mod N
\]
\[
e_1 = 0
\]
\[
A = \text{Mont}(A, A) = x^{1100} \cdot R \mod N
\]
\[
e_0 = 1 \rightarrow A = \text{Mont}(A, x') = x^{1101} \cdot R \mod N
\]
\[
\text{result} = \text{Mont}(A, 1) = x^{1101} \mod N
\]

Two initial values have to be calculated: $A$ and $x'$.

Example 2, an example exponent parsing with a window size of 2:
Calculation of $x^{11011} \mod N$ using exponent Parsing

\[
x[0] = \text{Mont}(x, R^2 \mod N) = x \cdot R \mod N
\]
\[
A = \text{Mont}(x[0], x[0]) = x^{10} \cdot R \mod N
\]
\[
x[1] = \text{Mont}(A, x[0]) = x^{11} \cdot R \mod N
\]
\[
A = x[1] = x^{11} \cdot R \mod N
\]
\[
e_2 = 0
\]
\[
A = \text{Mont}(A, A) = x^{110} \cdot R \mod N
\]
\[
e_0 = 1 \rightarrow A = \text{Mont}(A, x[1]) = x^{1100} \cdot R \mod N
\]
\[
\text{result} = \text{Mont}(A, 1) = x^{11011} \mod N
\]
Appendix E. Coprocessor instructions

E.1 COP:

31..28 cond  condition
27..24  1110
23..20 opcode_1  operand code, see also the command register
19..16 CRn  first source register
15..12 CRd  destination register
11..8 cp_num  coprocessor number
7..5 opcode_2  operand code, (second command register possible)
 4  0
 3..0 CRm  second source register

The commands (bits 23...20) that are executed in the PIP2025 are as following:
- Reset  000x
- MMM  001x
- MMMNEXT  011x
- EXP  100x

x = 0:  N register is internal, modulus has 1024 bits or less.
x = 1:  N operand is addressed using N Base register.

E.2 LOC:

31..28 cond  condition
27..25  110
24  P

Pre-indexing (1), offset is added to the base register, the result is used as the address
Post-indexing (0), the base register is used for the address, the offset is added to the base register and written back to the base register (because W will equal 1, see below)

23  U
Add (1) or Subtract (0) the offset to/from the base register for addresses
22  N  coprocessor-dependent
21  W  calculated address will be written back to the base register
If P = 0 then W must be equal to 1 or the result is unpredictable

20  1
19..16 Rn  specifies the base register
15..12 CRd  destination register
11..8 cp_num  coprocessor number
7..0 offset  8-bit word offset

E.3 MCR:

31..28 cond  condition
27..24  1110

Implementing Cryptographic Solutions in a Modular SmartCard core
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>23..21</td>
<td><code>opcode_1</code></td>
</tr>
<tr>
<td>20</td>
<td><code>0</code></td>
</tr>
<tr>
<td>19..16</td>
<td><code>CRn</code></td>
</tr>
<tr>
<td>15..12</td>
<td><code>Rd</code></td>
</tr>
<tr>
<td>11..8</td>
<td><code>cp_num</code></td>
</tr>
<tr>
<td>7..5</td>
<td><code>opcode_2</code></td>
</tr>
<tr>
<td>4</td>
<td><code>1</code></td>
</tr>
<tr>
<td>3..0</td>
<td><code>CRm</code></td>
</tr>
</tbody>
</table>

**E.4 MRC:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31..28</td>
<td><code>cond</code></td>
</tr>
<tr>
<td>27..24</td>
<td><code>1110</code></td>
</tr>
<tr>
<td>23..21</td>
<td><code>opcode_1</code></td>
</tr>
<tr>
<td>20</td>
<td><code>1</code></td>
</tr>
<tr>
<td>19..16</td>
<td><code>CRn</code></td>
</tr>
<tr>
<td>15..12</td>
<td><code>Rd</code></td>
</tr>
<tr>
<td>11..8</td>
<td><code>cp_num</code></td>
</tr>
<tr>
<td>7..5</td>
<td><code>opcode_2</code></td>
</tr>
<tr>
<td>4</td>
<td><code>1</code></td>
</tr>
<tr>
<td>3..0</td>
<td><code>CRm</code></td>
</tr>
</tbody>
</table>

**E.5 STC:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31..28</td>
<td><code>Cond</code></td>
</tr>
<tr>
<td>27..25</td>
<td><code>110</code></td>
</tr>
<tr>
<td>24</td>
<td><code>P</code></td>
</tr>
<tr>
<td>23</td>
<td><code>U</code></td>
</tr>
<tr>
<td>22</td>
<td><code>N</code></td>
</tr>
<tr>
<td>21</td>
<td><code>W</code></td>
</tr>
<tr>
<td>20</td>
<td><code>0</code></td>
</tr>
<tr>
<td>19..16</td>
<td><code>Rn</code></td>
</tr>
<tr>
<td>15..12</td>
<td><code>CRd</code></td>
</tr>
<tr>
<td>11..8</td>
<td><code>cp_num</code></td>
</tr>
<tr>
<td>7..0</td>
<td><code>offset</code></td>
</tr>
</tbody>
</table>

**Pre-indexing (1), offset is added to the base register, the result is used as the address**

**Post-indexing (0), the base register is used for the address, the offset is added to the base register and written back to the base register (because W will equal 1, see below)**

**Add (1) or Subtract (0) the offset to/from the base register for addresses**

**coprocessor-dependent**

**calculated address will be written back to the base register**

**If P =0 then W must be equal to 1 or the result is unpredictable**
Appendix F. Coprocessor instructions timetable

Coprocessor instructions

LDC-instruction for PIP
nCPI
CPA
CPB
ADDRBUS
DATABUS

CDP-instruction for PIP - e.g. MMM instruction - Memory activity controlled by PIP
nCPI
CPA
CPB
ADDRBUS
DATABUS

Interrupt from the CPU, the PIP freezes and continues after the NCPI gets active again
nCPI
CPA
CPB
ADDRBUS
DATABUS

Copyright © SafeNet. All rights reserved.
Appendix G. Class diagram of the PIP2025 software model

Class structure - PIP2025 software simulation (inheritance)
Class structure - PIP2025 software simulation (relations)
Appendix I. Architecture PE’s

First PE, termination PE and logic before and after systolic array
Appendix J. Software model

The forms that are visible by the customer for a simulation are:
- PIP2025 Level 0 (or 1)
- PIP2025 coprocessor
- CPU processor
- Memory
- Trace
- Clock

The forms that are visible with internal use for a simulation are:
- PIP2025 Level 3 (or 2)
- PIP2025 coprocessor
- PIPSC100
- PIPControl
- PIPM1M
- PIPRegisters
- CPU processor
- Memory
- Clock
- Trace
- Config

These forms are shown in the next paragraphs.

J.1 Customer

![Diagram of PIP2025 Level 1](image.png)
### Configuration Registers
- FIFO 32 x 32
- Processing Elements 1 ... n-1
- PE1
- PEN
- Data Register
- BUS Interface

### CPU
**Instruction encode**
- Condition: 0000 31.28
- Opcode_1: 0000 23.20
- CRn: 0000 19.16
- CRd: 0000 15.12
- cp_num: 0101 7.5
- Opcode_2: 000 4.0
- CRm: 0000 3.0

### Memory
**Data IN/OUT**
- Load Memory
- RW: xb
- nMReq: xb

### Trace
**Instruction**
- MCR
- CDP
- LDC
- STC
- MRC

**Command**
- Reset
- MMM
- MMMNext
- EXP

---

Appendices

Copyright © SafeNet. All rights reserved.
Implementing Cryptographic Solutions in a Modualr SmartCard core
**Configuration**

- Instruction: 1M
  - CR

**Legend Failure 0FIFO (33x32) N·R egister (32x32)**

- Command: Command
  - 0: N_acc
  - 1: B pointer
  - 2: B counter
  - 3: T base
  - 4: Y base
  - 5: N base
  - 6: X pointer
  - 7: n_digits
  - 8: n_xdigits
  - 9: n_passes
  - 10: n_exparra
  - 11: n_mode
  - 12: status

**View Changes**
J.3 Menu structure of the smartcard simulation

The structure of the menu of the SmartCard:

- **File**
  - Open Program
  - Close Program
  - Select Tracefile
  - Exit
- **Run**
  - Run Program
  - Show Data Changes
  - Trace File
- **Help**
  - Help Topic
  - About
- **View**
  - Clock
  - CPU
  - PIP
    - PIP 2025
    - PIP Control
    - PIP MMM
    - PIP Registers
    - PIP SC100
  - Memory
  - Trace
  - Config

The following forms are the same as the customer levels.
- CPU
- Memory
- Clock
- Trace
J.4 Program.prg example
MEM Load Memory.mem
MCR 0x00 0x00000001
MCR 0x01 0x00000025
MCR 0x02 0x000000be
MCR 0x03 0x00000060
MCR 0x04 0x00000000
MCR 0x05 0x00000040
MCR 0x06 0x00000000
MCR 0x07 0x00000006
MCR 0x08 0x00000006
MCR 0x09 0x00000007
MCR 0x0a 0x00000003
MCR 0x0b 0x00000000
LDC 0x00000040
CDP MMM
MOV 0x00000000 0x000000a0
MCR 0x03 0x000000a0
CDP MMM
MCR 0x03 0x00000000
MCR 0x04 0x000000a0
CDP MMNExt
MCR 0x04 0x000000a7
CDP MMNExt
MOV 0x000000a0 0x00000000
MCR 0x03 0x000000a0
MCR 0x04 0x00000000
CDP EXP
MCR 0x03 0x00000090
MCR 0x08 0x00000002
CDP MMM
MEM Result 0x00000080
MEM Store Store.mem

J.5 Memory.mem example
0x00000000 0xdddddddd
0x00000001 0xaaaaaa
0x00000002 0x55555555
0x00000003 0x55555555
0x00000004 0xaaaaaa
0x00000005 0xdddddddd
0x00000020 0xffffffff
0x00000021 0xffffffff
0x00000022 0xffffffff
0x00000023 0xffffffff
0x00000024 0xffffffff
0x00000025 0xffffffff
0x00000040 0xffffffff
0x00000041 0x66666666
0x00000042 0x00000000
0x00000043 0x00000000
0x00000044 0x66666666
0x00000045 0xffffffff
0x00000060 0xaea2f357
0x00000061 0x67765b4
0x00000062 0x23d62901
0x00000063 0x1811efc3
0x00000064 0xc7207d41
0x00000065 0x7b115321
0x00000080 0x5ca010a7
0x00000081 0xe7e7489f
0x00000082 0x84bb8764
0x00000083 0xbf9b3b42
0x00000084 0x62ec2c37
0x00000085 0x9c1c062
0x00000090 0x00000001
J.6 Trace.trc example

Begin Program

Total = 0

Init
nCPI = 1
CPA = 1
CPB = 1
nMRequest = 1

Begin Instruction

Clock = 0
Total = 1
Total = 2
Databus = 0x0e007510

Decode: MCR instruction
Clock = 1
Clock = 2
nCPI = 0

Execute the instruction
Databus = 0x00000001
CPA = 0
CPB = 0
Clock = 3
n_ydigits = 0x01
CPA = 1
CPB = 1

Execute ENDS
nCPI = 1

End Instruction

Total = 5

Running time = 0:0:0.36 seconds

Real time (20 MHz): 0.00025 milliseconds

End Program
Appendix K. Simulation level of internal and customer model

K.1 Simulation Level 0/1 PIP2025 - customer

Visible on monitor:
- Toplevel structure of the SmartCard with division in CPU, Memory and the PIP2025 modules
- The Control-lines between CPU, Memory and the PIP2025.
- The Databus and Addressbus.
- A separate form displays the Clock with the counter Total cycles of all instructions and a counter Clock for each instruction that is running.
- The status of the control-signals (i.e. CPA, CPB and nCPI).
- The data and addresses on the databus or addressbus in hexadecimal values.
- It is possible to make visible the layout of the PIP2025, CPU and Memory

Visible on trace-file:
- The beginning and ending of a set of instructions ("Begin Program", "End Program").
- The beginning and ending of an instruction ("Begin Instruction", "End Instruction").
- The positive clock edges
- Every change of the control-signals (i.e. CPA, CPB and nCPI).
- Which instruction is executing (hex-format).
- Every data and addresses put on the databus or addressbus.
- Which register will be read/write by the CPU (i.e. n_ydigits)
- Which N-register will be loaded or stored.

Simulation:
- A program simulation in coprocessor instructions that can be read from a text file.
- An pre-programmed instruction:
  - MCR: write a value to a coprocessor register, register-number as well as the value are adaptable
  - MRC: read a value from a coprocessor register
  - LDC: load N-register (#words is n_ydigits)
  - STC: store N-register (#words is n_ydigits)
  - CDP: execute one of the four coprocessor commands

K.2 Simulation Level 2/3 PIP2025 - internal

For the SafeNet internal simulation program extra features will be available. Except all the points mentioned above in the customer level, the following modules, signals, registers, etc. are accessible.

Visible on the monitor
All points from the customer version.
- Lower level structures from PIP to PE level
- All the registers
  - Values of the twelve accessible registers for the CPU in the interface module
  - Values stored in the N-register
  - Values stored in the X-register
Continuous monitoring of all stored values in FIFO (R-register), first (next value that is read) and last stored values are indicated.

- Values of the registers in the systolic array after every clock step
- The status of all the signals after every clock cycle on every level form PIP to PE (not inside the PE's), data values are hexadecimal

**Visible in the trace file**

All points from the customer version.

- All the rising clock edges during the execution of a coprocessor command
- Per cycle all the signal and data changes on every level (not inside the PE's), data values are hexadecimal

**Simulation**

The simulations are similar to the possible simulations in the customer's version, the executed simulation depends on the read file.
Appendix L. Performance figures

In the table are the performance figures shown for three interesting configurations. The first one is the 4 PE solution with $\alpha = 8$ and $\beta = 32$. This configuration is used for the first implementation of the PIP2025. The reason for that is the excellent performance for $1024 \times 1024$ bit and the gate count in comparison with the third option, that has almost the same performance for values smaller than $1024$-bits. The third configuration is a 8 PE solution with $\alpha = 8$ and $\beta = 16$. The second configuration has less performance but the number of gates for this solution is extremely small in comparison with the others. In this case the number of PE’s is 4, $\alpha = 8$ and $\beta = 16$.

<table>
<thead>
<tr>
<th>Max # Bits</th>
<th>#Odd</th>
<th>64</th>
<th>192</th>
<th>512</th>
<th>512</th>
<th>1024</th>
<th>1024</th>
<th>1024</th>
<th>2048</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td># Bits Exp</td>
<td>powers</td>
<td>64</td>
<td>192</td>
<td>192</td>
<td>512</td>
<td>192</td>
<td>512</td>
<td>1024</td>
<td>512</td>
<td>2048</td>
</tr>
<tr>
<td>Worst Case</td>
<td>1</td>
<td>4025</td>
<td>28093</td>
<td>126877</td>
<td>339997</td>
<td>447447</td>
<td>1193047</td>
<td>2386007</td>
<td>8796213</td>
<td>35184693</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3140</td>
<td>21318</td>
<td>96988</td>
<td>255082</td>
<td>339116</td>
<td>898316</td>
<td>1793104</td>
<td>6622957</td>
<td>26414317</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3174</td>
<td>21394</td>
<td>97324</td>
<td>257164</td>
<td>340284</td>
<td>899484</td>
<td>1794272</td>
<td>6631550</td>
<td>26422910</td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
<td>3019</td>
<td>21070</td>
<td>95158</td>
<td>254998</td>
<td>335585</td>
<td>894785</td>
<td>1789505</td>
<td>6597160</td>
<td>26386520</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2684</td>
<td>18731</td>
<td>84595</td>
<td>226693</td>
<td>298335</td>
<td>795464</td>
<td>1590870</td>
<td>5864875</td>
<td>23459394</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2518</td>
<td>17572</td>
<td>79362</td>
<td>212668</td>
<td>279878</td>
<td>746251</td>
<td>1492447</td>
<td>5502031</td>
<td>22008026</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Max # Bits</th>
<th>#Odd</th>
<th>192</th>
<th>1024</th>
<th>2048</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td># Bits Exp</td>
<td>powers</td>
<td>192</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td>Worst Case</td>
<td>1</td>
<td>46909</td>
<td>2308183</td>
<td>4616279</td>
<td>35184693</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>35577</td>
<td>1737935</td>
<td>3469007</td>
<td>26414317</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>35702</td>
<td>1740192</td>
<td>3471264</td>
<td>26422910</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
<td>35182</td>
<td>1731137</td>
<td>3462209</td>
<td>26388520</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>31277</td>
<td>1538981</td>
<td>3077904</td>
<td>23459394</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>29342</td>
<td>1443768</td>
<td>2887482</td>
<td>22008026</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Max # Bits</th>
<th>#Odd</th>
<th>192</th>
<th>1024</th>
<th>2048</th>
<th>1024</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td># Bits Exp</td>
<td>powers</td>
<td>192</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Worst Case</td>
<td>1</td>
<td>33085</td>
<td>1232983</td>
<td>2465879</td>
<td>18169909</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>25101</td>
<td>928385</td>
<td>1853057</td>
<td>13640767</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>25190</td>
<td>929592</td>
<td>1854264</td>
<td>13645206</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
<td>24814</td>
<td>924737</td>
<td>1849409</td>
<td>13627432</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>22060</td>
<td>822091</td>
<td>1644125</td>
<td>12114787</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>20695</td>
<td>771231</td>
<td>1542407</td>
<td>11365278</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table L-1. Performance figures of the PIP2025 software model (two configurations)
For visualization of the advantage using extra odd powers the differences for the implemented configuration (4PE’s, $\alpha = 8$ and $\beta = 32$) are combined in the next figures. The first figure shows the reduction with odd powers in the worst case scenario. The second figure shows the reduction using odd powers in an average case.

Figure L-1, Performance of PIP2025, average values for a full exponentiation

Figure L-2, Performance of PIP2025, average values for a full exponentiation
The differences for the three mentioned configurations are visualized and compared in the following two figures. The number of PE’s and Beta are adapted in the configurations. Per configuration the worst case scenario and the averages both with one and three odd-powers are simulated and compared. The used test vectors are 1024- and 2048-bit.

![Different configurations for 1024-bits](image)

Figure L-3, Comparison of different configurations with 1024-bit vectors

![Different configurations for 2048-bit](image)

Figure L-4, Comparison of different configurations with 2048-bit vectors
### Appendix M. Example of a generated VHDL command-file

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITN NAcc0 WRITEH 01</td>
<td>-- BITN FirstD CHECKH 00</td>
</tr>
<tr>
<td>BITN Din WRITEH 00000200</td>
<td>-- BITN FirstT CHECKH 0000aa05</td>
</tr>
<tr>
<td>BITN NRegIn WRITEH ffffffff</td>
<td>-- BITN SecD CHECKH b0</td>
</tr>
<tr>
<td>BITN FifoIn WRITEH 00000000</td>
<td>-- BITN SecT CHECKH 0fffd2abbd</td>
</tr>
<tr>
<td>BITN CtrlVector WRITEH 0005</td>
<td>-- BITN MainD CHECKH bd</td>
</tr>
<tr>
<td>SYS WAITTIME 1</td>
<td>-- BITN MainT CHECKH 003000b54</td>
</tr>
<tr>
<td>BITN ExpResult CHECKH ed634073</td>
<td>-- BITN LastD CHECKH 54</td>
</tr>
<tr>
<td>-- BITN FirstD CHECKH 00</td>
<td>-- BITN LastT CHECKH 0092e582d</td>
</tr>
<tr>
<td>-- BITN FirstT CHECKH 605f5b58</td>
<td></td>
</tr>
<tr>
<td>-- BITN SecD CHECKH 03</td>
<td></td>
</tr>
<tr>
<td>-- BITN SecT CHECKH 055d253ff</td>
<td></td>
</tr>
<tr>
<td>-- BITN MainD CHECKH f4</td>
<td></td>
</tr>
<tr>
<td>-- BITN MainT CHECKH 00bfa557</td>
<td></td>
</tr>
<tr>
<td>-- BITN LastD CHECKH 57</td>
<td></td>
</tr>
<tr>
<td>-- BITN LastT CHECKH 0c21f5b52</td>
<td></td>
</tr>
<tr>
<td>SYS WAITCLK 2</td>
<td></td>
</tr>
<tr>
<td>BITN NAcc0 WRITEH 01</td>
<td>BITS ExpResult CHECKH 5d2e582d</td>
</tr>
<tr>
<td>BITS ExpResult CHECKH ed634073</td>
<td></td>
</tr>
<tr>
<td>BITS ExpResult CHECKH ed634073</td>
<td></td>
</tr>
<tr>
<td>SYS WAITCLK 2</td>
<td></td>
</tr>
<tr>
<td>BITS ExpResult CHECKH 5d2e582d</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>BITS ExpResult CHECKH ed634073</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implementing Cryptographic Solutions in a Modular SmartCard core

Copyright © SafeNet. All rights reserved.
Appendix N. Class diagrams of elliptic curve representations

N.1 Binary field with affine coordinates

![Class diagram of elliptic curve representations](image-url)
N.2 Binary field with projective coordinates

Diagram showing the relationships and operations involved in binary field encryption and decryption, with scalar multiplication, point adding, point doubling, inversion, multiplication, squaring, and addition operations.
N.3 Binary field projective coordinates using Montgomery

Diagram showing operations and their corresponding number of bits.
N.4 Prime field with affine coordinates

encryption / decryption

scalar mult.

Point Adding

Point Doubling

#ones

#bits

multiplication

squaring

inversion

addition / subtraction

Copyright © SafeNet. All rights reserved.
N.5 Prime field with projective coordinates

encryption / decryption

scalar mult.

Point Adding

#ones

Point Doubling

#bits

inversion

multiplication

addition / subtraction

Page 36

Appendices

Copyright © SafeNet. All rights reserved.
Appendix O. Calculation sequence of different EC representations

0.1 Calculations for BINARY field – AFFINE coordinates

Elliptic Adding

The affine form of the adding formula on the curve $y^2 = x^3 + ax + b$ over GF($2^m$), is

$$(X_p, Y_p) + (X_q, Y_q) = (X_r, Y_r).$$

**Input:** a field of $2^m$ elements; the field elements a en b defining a curve E over GF($2^m$); affine coordinates $(X_p, Y_p)$ and $(X_q, Y_q)$ for points P and Q on E.

**Output:** affine coordinates $(X_r, Y_r)$ for the point $R = P + Q$, unless $P = Q$. In this case, the triplet $(0, 0, 0)$ is returned.

**NOTE:** the italic-lines are extra for executing the calculation using 4 registers or against timing/power - attacks

Algorithm O-1, Point-adding with affine coordinates in a binary field

| T1 = x_q |
| T2 = y_q |
| T3 = x_p |
| T1 = T1 + T3 = x_q - x_p = x_q + x_p |
| T4 = y_p |
| T2 = T2 + T4 = y_q - y_p |
| x_q = T1 |
| y_q = T2 |
| y_q = T2 / T1 |
| T4 = T2 / T1 = (y_q - y_p) / (x_q - x_p) |
| T1 = x_q |
| T2 = T2 + T4 |
| T1 = T2 + T1 |
| if a ≠ 0 |
| T2 = a |
| T1 = 2T |
| else |
| T1 = 2T |
| if b ≠ 0 |
| T2 = b |
| T3 = T3 + T1 |
| T2 = y_p |
| if y_p parallel with next calculation |
| y_p = (y_q - y_p) / (x_q - x_p) |

Elliptic Doubling

The affine form of the adding formula on the curve $y^2 = x^3 + ax + b$ over GF($2^m$), is

$$2 (X_q, Y_q) = (X_r, Y_r).$$

**Input:** a field of $2^m$ elements; the field elements a en b defining a curve E over GF($2^m$); affine coordinates $(X_q, Y_q)$ for point Q on E.

**Output:** affine coordinates $(X_r, Y_r)$ for the point $R = 2Q$.

**NOTE:** the italic-lines are extra for executing the calculation using 4 registers or against timing/power attacks.

Implementing Cryptographic Solutions in a Modular SmartCard Core

Copyright © SafeNet. All rights reserved.
Algorithm O-2, Point-doubling with affine coordinates in a binary field

\[ T_1 = x_q \]
\[ T_2 = y_q \]
\[ T_3 = x_p \]
\[ T_4 = T_2 + T_1 \]
\[ T_5 = y_p \]
\[ x_q = T_7 \]
\[ y_q = T_2 \]
\[ T_3 = f \]
\[ T_4 = T_2 \times T_3 \]
\[ T_5 = T_2 + T_1 \]
\[ y_q = T_2 \]
\[ T_3 = T_2 + T_1 \]
\[ T_4 = T_2 \times T_3 \]
\[ T_5 = T_2 + T_1 \]
\[ if \ a \neq 0 \]
\[ T_6 = a \]
\[ T_7 = T_4 + T_1 \]
\[ else \]
\[ T_1 = \frac{y_q}{x_q} \]
\[ T_2 = x_q \]
\[ T_3 = T_4 + T_2 \]
\[ T_4 = y_q \]
\[ T_5 = T_2 + T_1 \]
\[ T_6 = x_q \]
\[ T_7 = x_r \]
\[ if \ Z_l \neq 1 \]
\[ T_8 = x_q \]
\[ T_9 = x_r \]
\[ T_1 = \frac{y_q}{x_q} \]
\[ T_2 = (Y_0 + Y_0) / \prod \]
\[ T_3 = (Y_0 + Y_0) \times (y_q / x_q) \]
\[ T_4 = (Y_0 + Y_0) \times (y_q / x_q) \times x_q \]
\[ T_5 = (Y_0 + Y_0) \times (y_q / x_q) \times x_q \times y_q \]

**O.2 Calculations for BINARY field – PROJECTIVE coordinates**

**Elliptic Adding**

The projective form of the adding formula on the curve \( y^2 = x^3 + ax + b \) over \( GF(2^m) \), is

\[ (X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2). \]

**Input:** a field of \( 2^n \) elements; the field elements \( a \) en \( b \) defining a curve \( E \) over \( GF(2^m) \); projective coordinates \( (X_0, Y_0, Z_0) \) and \( (X_1, Y_1, Z_1) \) for points \( P_0 \) and \( P_1 \) on \( E \), where \( Z_0 \) and \( Z_1 \) are nonzero.

**Output:** projective coordinates \( (X_2, Y_2, Z_2) \) for the point \( P_2 = P_0 + P_1 \), unless \( P_0 = P_1 \). In this case, the triplet \((0, 0, 0)\) is returned. (The triplet \((0, 0, 0)\) is not a valid projective point on the curve, but rather a marker indicating that routine Double should be used.)

1. \( T_1 \leftarrow X_0 \)
2. \( T_2 \leftarrow Y_0 \)
3. \( T_3 \leftarrow Z_0 \)
4. \( T_4 \leftarrow X_1 \)
5. \( T_5 \leftarrow Y_1 \)
6. \( T_6 \leftarrow X_0 \)
7. If \( a \neq 0 \) then
8. \( T_7 \leftarrow T_6^2 \)
9. \( T_8 \leftarrow T_7 \times T_7 \)
10. If \( Z_1 \neq 1 \) then
11. \( T_9 \leftarrow Z_1 \)
12. \( T_7 \leftarrow T_7 \times T_7 \)
13. \( T_4 \leftarrow T_4 + T_1 \)
14. \( T_6 \leftarrow \frac{y_q}{x_q} \)
15. \( T_5 \leftarrow \frac{y_q}{x_q} \times x_q \)
16. \( T_7 \leftarrow \frac{y_q}{x_q} \times x_q \times y_q \)
Elliptic Doubling

The projective form of the doubling formula on the curve $y^2 = x^3 + ax + b$ over $\text{GF}(2^m)$ uses, not the coefficient $b$, but rather the field element: $c = b^{2^{m-2}}$, computed from $b$ by $m - 2$ squarings.

\[ 2 (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2) \]

**Input:** a field of $2^m$ elements; the field elements $a$ and $c$ specifying a curve $E$ over $\text{GF}(2^m)$; projective coordinates $(X_1, Y_1, Z_1)$ for a point $P_1$ on $E$.

**Output:** projective coordinates $(X_2, Y_2, Z_2)$ for the point $P_2 = 2P_1$

1. $T_1 \leftarrow X_1$
2. $T_2 \leftarrow Y_1$
3. $T_3 \leftarrow Z_1$
4. $T_4 \leftarrow c$
5. If $T_1 = 0$ or $T_3 = 0$ then output $(1,1,0)$ and stop.
6. $T_5 \leftarrow T_2 \times T_3$
7. $T_6 \leftarrow T_5^2$
8. $T_7 \leftarrow T_1 \times T_4$
9. $T_8 \leftarrow T_3 \times T_4$
10. $T_9 \leftarrow T_2 + T_3$
11. $T_10 \leftarrow T_1 + T_4$
12. $T_11 \leftarrow T_2 \times T_4$
13. $T_12 \leftarrow T_4^2$
14. $T_13 \leftarrow T_4^2$
15. $T_14 \leftarrow T_1^2$
16. $T_15 \leftarrow T_1 \times T_4$
17. $T_16 \leftarrow T_2 \times T_4$
18. $T_17 \leftarrow T_1^2$
18. \( T_1 \leftarrow T_1 \times T_3 \)
19. \( T_2 \leftarrow T_1 + T_2 \)
20. \( T_3 \leftarrow T_4 \)
21. \( X_1 \leftarrow T_1 \)
22. \( Y_2 \leftarrow T_2 \)
23. \( Z_2 \leftarrow T_3 \)

\section*{0.3 Calculations for BINARY field – PROJECTIVE coordinates with MONTGOMERY}

**Elliptic adding**

The projective form of the adding formula using Montgomery is:

\[ (X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2), \]

also use \( X_p \): the original \( x \)-coordinate of \( P \),

Algorithm 0-3, Point-adding with projective coordinates in a binary field using Montgomery

\[
\begin{align*}
T_1 & \leftarrow X_0 \\
T_2 & \leftarrow Z_1 \\
T_3 & \leftarrow X_1 \\
T_4 & \leftarrow Z_0 \\
T_1 &= T_1 \times T_2 \\
T_2 &= T_3 \times T_4 \\
T_3 &= T_1 + T_2 \\
T_4 &= T_1 \times T_2 \\
T_2 &= T_3 \times T_3 \\
T_2 &= (X_0 \times Z_1 + X_1 \times Z_0)^2 = Z_2 \\
Z_2 &= T_2 \\
T_4 &= T_3 \times X_p \\
T_3 &= T_2 \times T_4 \\
T_3 &= T_1 + T_3 \\
X_2 &= T_3 \\
\end{align*}
\]

note: \( Y \) is not used in this Montgomery calculation

**Elliptic Doubling**

The projective form of the doubling formula using Montgomery is:

\[ 2(X_0, Y_0, Z_0) = (X_2, Y_2, Z_2), \]

Algorithm 0-4, Point-doubling with projective coordinates in a binary field using Montgomery

\[
\begin{align*}
T_1 & \leftarrow X_0 \\
T_1 &= T_1 \times T_1 \\
T_2 & \leftarrow Z_0 \\
T_2 &= T_2 \times T_2 \\
T_4 & \leftarrow T_1 \times T_2 \\
T_4 &= X_0^2 \times Z_0^2 = Z_2 \\
Z_2 &= T_2 \\
T_3 & \leftarrow b \\
T_1 &= T_1 \times T_1 \\
T_2 &= T_2 \times T_2 \\
T_2 &= b \times Z_0^4 \\
T_2 &= T_1 \times T_2 \\
X_2 &= X_0^4 + b \times Z_0^4 = X_2 \\
X_2 & \leftarrow T_3 \\
\end{align*}
\]

note: \( Y \) is not used in this Montgomery calculation

\section*{0.4 Calculations for PRIME field – AFFINE coordinates}

The sequence of this configuration is not worked out because it is clear that the number of gates necessary to have a reasonable performance is much larger than desired. Therefore this configuration is not chosen for implementation of elliptic curves. The reason for that worse performance or gate-
count is the inversion. Two inversions are calculated per point-adding or -doubling. The inversion takes a lot of gates in hardware or has a worst performance.

**Elliptic Adding**

\[
x_r = \left( \frac{y_q - y_p}{x_q - x_p} \right)^2 - x_p - x_q
\]

Equation A-O-1

\[
y_r = -y_p + \left( \frac{y_q - y_p}{x_q - x_p} \right)(x_p - x_r)
\]

Equation A-O-2

**Elliptic Doubling**

\[
x_r = \left( \frac{3x_p^2 + a}{2y_p} \right)^2 - 2x_p
\]

Equation A-O-3

\[
y_r = -y_p + \left( \frac{3x_p^2 + a}{2y_p} \right)(x_p - x_r)
\]

Equation A-O-4

### O.5 Calculations for PRIME field – PROJECTIVE coordinates

**Elliptic Adding**

The projective form of the adding formula on the curve \( y^2 = x^3 + ax + b \mod p \), is

\[(X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2), \]

**Input:** a modulus \( p \); the coefficients \( a \) and \( b \) defining a curve \( E \mod p \); projective coordinates \((X_0, Y_0, Z_0)\) and \((X_1, Y_1, Z_1)\) for points \( P_0 \) and \( P_1 \) on \( E \), where \( Z_0 \) and \( Z_1 \) are nonzero.

**Output:** projective coordinates \((X_2, Y_2, Z_2)\) for the point \( P_2 = P_0 + P_1 \), unless \( P_0 = P_1 \). In this case, the triplet \((0, 0, 0)\) is returned. (The triplet \((0, 0, 0)\) is not a valid projective point on the curve, but rather a marker indicating that routine Double should be used.)

1. \( T_1 \leftarrow X_0 \)
2. \( T_2 \leftarrow Y_0 \)
3. \( T_3 \leftarrow Z_0 \)
4. \( T_4 \leftarrow X_1 \)
5. \( T_5 \leftarrow Y_1 \)
6. If \( Z_1 \neq 1 \) then
   \[ T_6 \leftarrow Z_1 \]
   \[ T_7 \leftarrow T_6^2 \]
   \[ T_1 \leftarrow T_7 \times T_7 \]
   \[ T_7 \leftarrow T_6 \times T_7 \]
   \[ T_2 \leftarrow T_3 \times T_7 \]
   \[ = U_0 \text{ if } Z_1 = 1 \]
7. \( T_7 \leftarrow T_3^2 \)
8. \( T_8 \leftarrow T_4 \times T_7 \)
9. \( T_7 \leftarrow T_3 \times T_7 \)
10. \( T_5 \leftarrow T_5 \times T_7 \)
11. \( T_2 \leftarrow T_1 - T_4 \)
12. \( T_3 \leftarrow T_2 - T_5 \)
13. If \( T_4 = 0 \) then
    \[ \text{If } T_5 = 0 \text{ then output } (0,0,0) \text{ and stop} \]
Elliptic Doubling

The projective form of the doubling formula on the curve $y^2 = x^3 + ax + b$ modulo $p$ is

$$2 \ (X, Y, Z) = (X_2, Y_2, Z_2),$$

**Input:** a modulus $p$; the coefficients $a$ and $b$ defining a curve $E$ modulo $p$; projective coordinates $(X_1, Y_1, Z_1)$ for a point $P_1$ on $E$.

**Output:** projective coordinates $(X_2, Y_2, Z_2)$ for the point $P_2 = 2P_1$.

1. $T_1 \leftarrow X_1$
2. $T_2 \leftarrow Y_1$
3. $T_3 \leftarrow Z_1$
4. If $T_2 = 0$ or $T_3 = 0$ then output $(1, 1, 0)$ and stop.
5. If $a = p - 3$ then
   $$T_4 \leftarrow T_3^2$$
   $$T_5 \leftarrow T_1 - T_4$$
   $$T_4 \leftarrow T_1 + T_4$$
   $$T_3 \leftarrow T_4 \times T_5$$
   $$T_4 \leftarrow 3 \times T_3$$
   $$= M$$
   else
   $$T_4 \leftarrow a$$
   $$T_5 \leftarrow T_3^2$$
   $$T_5 \leftarrow T_5^2$$
   $$T_3 \leftarrow T_4 \times T_5$$
   $$T_4 \leftarrow T_1^2$$
   $$T_4 \leftarrow 3 \times T_4$$
   $$T_4 \leftarrow T_4 + T_5$$
   $$= M$$
6. $T_3 \leftarrow T_5 \times T_3$
7. $T_3 \leftarrow 2 \times T_3$
8. $T_3 \leftarrow T_2^2$
9. $T_3 \leftarrow T_5 \times T_2$
10. $T_3 \leftarrow 4 \times T_3$
11. $T_3 \leftarrow T_4^2$
12. $T_3 \leftarrow T_1 - 2 \times T_5$

Appendices

Page 42
P.1 Execution sequence to prevent timing and power attacks

In the following table the execution sequences for projective as well as affine coordinates are shown. An indication for the number of cycles for the operation is also made. This value is dependent on the number of bits of the reduction polynomial (m). The depth of the multiplier is one.

**PROJECTIVE COORDINATES**

<table>
<thead>
<tr>
<th>Adding</th>
<th>Doubling</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load: Z1</td>
<td>load: Z1</td>
<td>m/32</td>
</tr>
<tr>
<td>squaring and load: X0</td>
<td>load: Y1</td>
<td>m</td>
</tr>
<tr>
<td>multiplication</td>
<td>multiplication</td>
<td>m</td>
</tr>
<tr>
<td>multiplication and load: Y0</td>
<td>squaring and fake load</td>
<td>m</td>
</tr>
<tr>
<td>multiplication and load: Z0, store: U0</td>
<td>multiplication and load: X1, fake store</td>
<td>m</td>
</tr>
<tr>
<td>squaring and store: S0</td>
<td>multiplication and fake store</td>
<td>m</td>
</tr>
<tr>
<td>multiplication and load: Y1, load: X1</td>
<td>fake multiplication and 2x fake load</td>
<td>m</td>
</tr>
<tr>
<td>multiplication</td>
<td>fake multiplication</td>
<td>m</td>
</tr>
<tr>
<td>multiplication and load: U0 (or X0)</td>
<td>fake multiplication and fake load</td>
<td>m</td>
</tr>
<tr>
<td>subtraction</td>
<td>fake subtraction</td>
<td>1</td>
</tr>
<tr>
<td>load: S0 (or Y0)</td>
<td>fake load</td>
<td>m/32</td>
</tr>
<tr>
<td>subtraction</td>
<td>addition</td>
<td>1</td>
</tr>
<tr>
<td>subtraction</td>
<td>fake subtraction</td>
<td>1</td>
</tr>
<tr>
<td>subtraction</td>
<td>fake subtraction</td>
<td>1</td>
</tr>
<tr>
<td>store: R</td>
<td>fake store</td>
<td>m/32</td>
</tr>
<tr>
<td>store: M</td>
<td>fake store</td>
<td>m/32</td>
</tr>
<tr>
<td>load: Z1</td>
<td>fake load</td>
<td>m/32</td>
</tr>
<tr>
<td>multiplication</td>
<td>squaring</td>
<td>m</td>
</tr>
<tr>
<td>multiplication</td>
<td>squaring</td>
<td>m</td>
</tr>
<tr>
<td>squaring and store: Z2</td>
<td>multiplication and store Z2</td>
<td>m</td>
</tr>
<tr>
<td>multiplication and load: R</td>
<td>fake load, addition</td>
<td>m</td>
</tr>
<tr>
<td>multiplication</td>
<td>multiplication</td>
<td>m</td>
</tr>
<tr>
<td>squaring</td>
<td>multiplication</td>
<td>m</td>
</tr>
<tr>
<td>subtraction</td>
<td>fake subtraction</td>
<td>1</td>
</tr>
<tr>
<td>subtraction</td>
<td>fake subtraction</td>
<td>1</td>
</tr>
<tr>
<td>multiplication and store: X2, load: M</td>
<td>multiplication and store: X2, fake load</td>
<td>m</td>
</tr>
<tr>
<td>multiplication</td>
<td>fake multiplication</td>
<td>m</td>
</tr>
<tr>
<td>subtraction</td>
<td>addition</td>
<td>1</td>
</tr>
<tr>
<td>right shift 1</td>
<td>fake shift</td>
<td>1</td>
</tr>
<tr>
<td>store: Y2</td>
<td>store: Y2</td>
<td>m/32</td>
</tr>
</tbody>
</table>

**Table P-1, Sequence for projective point-addings and point-doublings**
It is also possible to use a Montgomery calculation, executing a scalar multiplication in binary field with projective coordinates. The sequence is than as following:

<table>
<thead>
<tr>
<th>Adding</th>
<th>cycles</th>
<th>Doubling</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load: Xq</td>
<td>m/32</td>
<td>load: Xq</td>
<td>m/32</td>
</tr>
<tr>
<td>load: Zp</td>
<td>m/32</td>
<td>multiplication and load: Zq</td>
<td>m</td>
</tr>
<tr>
<td>multiplication and load: Xp</td>
<td>m</td>
<td>multiplication and load: b</td>
<td>m</td>
</tr>
<tr>
<td>load: Zq</td>
<td>m/32</td>
<td>multiplication</td>
<td>m</td>
</tr>
<tr>
<td>multiplication</td>
<td>m</td>
<td>multiplication</td>
<td>m</td>
</tr>
<tr>
<td>addition</td>
<td>1</td>
<td>multiplication</td>
<td>m</td>
</tr>
<tr>
<td>multiplication</td>
<td>m</td>
<td>addition</td>
<td>1</td>
</tr>
<tr>
<td>multiplication and load: P</td>
<td>m</td>
<td>store: Zr</td>
<td>m/32</td>
</tr>
<tr>
<td>addition</td>
<td>1</td>
<td>store: Xr</td>
<td>m/32</td>
</tr>
<tr>
<td>store: Zr</td>
<td>m/32</td>
<td>store: Zr</td>
<td>m/32</td>
</tr>
<tr>
<td>store: Xr</td>
<td>m/32</td>
<td>Total</td>
<td>6 3/32m + 1</td>
</tr>
<tr>
<td>Total</td>
<td>5 5/32m + 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table P-2, Sequence for point-adding and point-doubling (montgomery)

**AFFINE COORDINATES**

<table>
<thead>
<tr>
<th>Adding</th>
<th>Doubling</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load: Xp</td>
<td>fake load</td>
<td>m/32</td>
</tr>
<tr>
<td>addition</td>
<td>fake addition</td>
<td>1</td>
</tr>
<tr>
<td>load: Yp</td>
<td>fake load</td>
<td>m/32</td>
</tr>
<tr>
<td>addition</td>
<td>fake addition</td>
<td>1</td>
</tr>
<tr>
<td>store: Xq + Xp</td>
<td>store: Xq</td>
<td>m/32</td>
</tr>
<tr>
<td>fake store</td>
<td>store: Yq</td>
<td>m/32</td>
</tr>
<tr>
<td>load: f</td>
<td>load: f</td>
<td>m/32</td>
</tr>
<tr>
<td>inversion</td>
<td>inversion</td>
<td>6-m</td>
</tr>
<tr>
<td>load: Xp</td>
<td>load Xq</td>
<td>m/32</td>
</tr>
<tr>
<td>fake addition</td>
<td>addition</td>
<td>1</td>
</tr>
<tr>
<td>multiplication and load: (Xq + Xp)</td>
<td>multiplication and fake load</td>
<td>m</td>
</tr>
<tr>
<td>addition</td>
<td>addition</td>
<td>1</td>
</tr>
<tr>
<td>addition</td>
<td>fake addition</td>
<td>1</td>
</tr>
<tr>
<td>if a != 0 load: a</td>
<td>if a != 0 load: a</td>
<td>m/32</td>
</tr>
<tr>
<td>if a != 0 addition</td>
<td>if a != 0 addition</td>
<td>1</td>
</tr>
<tr>
<td>addition</td>
<td>addition</td>
<td>1</td>
</tr>
<tr>
<td>multiplication and load: Yp</td>
<td>multiplication and load: Yq</td>
<td>m</td>
</tr>
<tr>
<td>addition</td>
<td>addition</td>
<td>1</td>
</tr>
<tr>
<td>addition</td>
<td>addition</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>8 3/32 m + 9</td>
<td></td>
</tr>
</tbody>
</table>

Table P-3, Sequence for affine point-addings and point-doublings
P.2 Formulas for the number of cycles with and without prevention

The number of cycles for a scalar multiplication is dependent from the length of scalar and the number of cycles necessary for an inversion. In the tables above an inversion takes 6m cycles. In the equations the depth of the multiplier is taken one. This will probably be implemented, because the number of gates has to be as less as possible. In the equations m is the number of bits, \( W_H(s_m) \) is the Hemming weight of the scalar.

**Projective coordinates, without power and timing attack prevention:**

\[
\text{# Cycles} = 4 \cdot \text{#Cycl.Inv.} + m \cdot \left[ 8 \cdot \frac{3}{32} m + 4 \right] + W_H(s_m) \cdot \left[ 16 \cdot \frac{7}{32} m + 8 \right]
\]

Equation P-1

**Projective coordinates, with power and timing attack prevention:**

\[
\text{# Cycles} = 4 \cdot \text{#Cycl.Inv.} + m + W_H(s_m) \cdot \left[ 16 \cdot \frac{7}{32} m + 8 \right]
\]

Equation P-2

**Projective coordinates, using Montgomery:**

\[
\text{# Cycles} = 2 \cdot (\text{#Cycl.Inv.} + m) + m \cdot \left[ 1 \frac{1}{4} m + 3 \right]
\]

Equation P-3

**Affine coordinates, without power and timing attack prevention:**

\[
\text{# Cycles} = \left[ m + W_H(s_m) \right] \cdot \left[ 2 \frac{1}{8} m + \#\text{Cycl.Inv.} + 7 \frac{1}{2} \right]
\]

Equation P-4

**Affine coordinates, with power and timing attack prevention:**

\[
\text{# Cycles} = \left[ m + W_H(s_m) \right] \cdot \left[ 2 \frac{7}{32} m + \#\text{Cycl.Inv.} + 9 \right]
\]

Equation P-5
Appendix Q. Area and performance estimations for different ECC implementations

The estimations are for a reduction polynomial of 210-bits other bit sizes give similar results. The totals for the three specified configurations of other polynomial lengths are summarized in the report itself. The first part contains all investigated implementations. In the second part are the three important implementations are specified. first the cycles and gates for the basic operations are summarized. then the area and performance figures for the control module are specified. Finally the figures for the registers and the interface are shown.

Configuration

<table>
<thead>
<tr>
<th>Basis</th>
<th>Coordinates</th>
<th>Multiplier</th>
<th>Gates</th>
<th>Cycles</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Affine</td>
<td>Massey Omura</td>
<td>27945</td>
<td>747321</td>
<td>Not scalable</td>
</tr>
<tr>
<td></td>
<td>Projective</td>
<td>Massey Omura</td>
<td>31105</td>
<td>525021</td>
<td>Not scalable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Matrix-Achtig</td>
<td>45070</td>
<td>2512242</td>
<td></td>
</tr>
<tr>
<td>Polynomial</td>
<td>Affine</td>
<td>Shift-and-Add Mult.</td>
<td>31335</td>
<td>532243</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Combined Add/Mult./Sq.</td>
<td>30200</td>
<td>532243</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Projective</td>
<td>Combined Add/Mult./Sq.</td>
<td>29740</td>
<td>844947</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Matrix Mult./Sq.</td>
<td>32595</td>
<td>531929</td>
<td></td>
</tr>
<tr>
<td>Montgomery P.</td>
<td>Projective</td>
<td>Combined Add/Mult./Sq.</td>
<td>29740</td>
<td>469142</td>
<td></td>
</tr>
</tbody>
</table>

The three most important implementations specified, these are all polynomial based.

### Coordinates

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Affine</th>
<th>Projective</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Combined Add/Mult./Sq.</td>
<td>Combined Add/Mult./Sq.</td>
</tr>
<tr>
<td>Addition</td>
<td>#</td>
<td>#Cycles</td>
</tr>
<tr>
<td>Multiply</td>
<td>2199</td>
<td>1</td>
</tr>
<tr>
<td>1bit Reg.</td>
<td>628</td>
<td>210</td>
</tr>
<tr>
<td>Squaring</td>
<td>3110</td>
<td>210</td>
</tr>
<tr>
<td>Muxing</td>
<td>1260</td>
<td>0</td>
</tr>
<tr>
<td>Conversion</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inversion</td>
<td>314</td>
<td>1050</td>
</tr>
<tr>
<td>Total</td>
<td>x</td>
<td>529719</td>
</tr>
</tbody>
</table>

### Standard

<table>
<thead>
<tr>
<th>Montgomery</th>
<th>#</th>
<th>#Cycles</th>
<th>#</th>
<th>#Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>1</td>
<td>2</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>Multiply</td>
<td>105</td>
<td>8</td>
<td>1500</td>
<td>105</td>
</tr>
<tr>
<td>Doubling</td>
<td>209</td>
<td>8</td>
<td>1500</td>
<td>210</td>
</tr>
<tr>
<td>other</td>
<td>x</td>
<td>10</td>
<td>1000</td>
<td>x</td>
</tr>
<tr>
<td>Total</td>
<td>x</td>
<td>2524</td>
<td>4500</td>
<td>x</td>
</tr>
</tbody>
</table>

### Addresses

<table>
<thead>
<tr>
<th>polynomial f</th>
<th>#</th>
<th>#Cycles</th>
<th>#</th>
<th>#Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>other 1-bit</td>
<td>19</td>
<td>x</td>
<td>190</td>
<td>19</td>
</tr>
<tr>
<td>other m-bit</td>
<td>4</td>
<td>x</td>
<td>8400</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>x</td>
<td>x</td>
<td>12930</td>
<td>x</td>
</tr>
</tbody>
</table>

### Interface

<table>
<thead>
<tr>
<th>#</th>
<th>#Cycles</th>
<th>#</th>
<th>#Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>4000</td>
<td>x</td>
</tr>
<tr>
<td>Total</td>
<td>x</td>
<td>532243</td>
<td>30200</td>
</tr>
</tbody>
</table>

Table Q-1. Comparison of different ECC implementations, interesting ones are specified.
Appendix R. Elliptic Curve Digital Signature Algorithm

First an elliptic curve $E$ defined over $\text{GF}(2^m)$ with large group of order $n$ and a point $P$ of larger order is selected and made public to all users. Then, each party uses the following key generation primitive to generate the individual public and private key pairs. Furthermore, for each transaction the signature and verification primitives are used. The steps for the Elliptic Curve Digital Signature Algorithm (ECDSA) are written below.

Key Generation
The user $A$ follows these steps:
2. Compute $Q = d \cdot P$.
3. The public key of the user $A$ is $(E, P, n, Q)$ and the private key is $d$.

Signature Generation
The user $A$ signs the message using the following steps:
2. Compute $(x_1, y_1) = k \cdot P$ and $r = x_1 \mod n$, if $x_1 \in \text{GF}(2^m)$, it is assumed that $x_1$ is represented as a binary number. If $r = 0$ then go to step 1.
3. Compute $k^{-1} \mod n$.
4. Compute $s = k^{-1} \cdot (H(\text{message}) + d \cdot r) \mod n$. Here $H$ is the secure hash algorithm SHA. If $s = 0$ go to step 1.
5. The signature for the message is the pair of integers $(r, s)$.

Signature Verification
The user $B$ verifies $A$'s signature $(r, s)$ on the message by applying the following steps:
1. Compute $c = s^{-1} \mod n$ and $H(\text{message})$
2. Compute $u_1 = H(\text{message}) \cdot c \mod n$ and $u_2 = r \cdot c \mod n$
3. Compute $(x_0, y_0) = u_1 \cdot P + u_2 \cdot Q$ and $v = x_0 \mod n$
4. Accept the signature if $v = r$
### Appendix S. High-level control signals of the ECC-implementation

In the table are the mentioned signals the output-signals of the corresponding modules.

#### Control module

<table>
<thead>
<tr>
<th>Name</th>
<th>#bits</th>
<th>value</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetS</td>
<td>1</td>
<td>0</td>
<td>no S-word is needed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>a S-word is needed</td>
</tr>
<tr>
<td>LoadS</td>
<td>1</td>
<td>0</td>
<td>no S-word has to be loaded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>load the S-word into the register</td>
</tr>
<tr>
<td>Finish</td>
<td>1</td>
<td>0</td>
<td>the called CDP-instruction is busy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>the called CDP-instruction is finished</td>
</tr>
<tr>
<td>Scounter</td>
<td>5</td>
<td></td>
<td>counter of the bits pointed to S-word</td>
</tr>
<tr>
<td>Address(13:0)</td>
<td></td>
<td></td>
<td>the address for loading a S-word for the scalar</td>
</tr>
<tr>
<td>Ain-pre</td>
<td>2</td>
<td>0</td>
<td>pointing to TempRegisters (signal passes basic module)</td>
</tr>
<tr>
<td>Bin-pre</td>
<td>2</td>
<td>0</td>
<td>pointing to TempRegisters (signal passes basic module)</td>
</tr>
<tr>
<td>Rout-pre</td>
<td>2</td>
<td></td>
<td>pointing to TempRegisters (signal passes basic module)</td>
</tr>
<tr>
<td>IA-pre</td>
<td>1</td>
<td></td>
<td>data available for registers (signal passes basic module)</td>
</tr>
<tr>
<td>MAl-pre</td>
<td>2</td>
<td>00</td>
<td>perform an addition (signal passes basic module)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01</td>
<td>perform a multiplication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>perform an inversion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>do nothing</td>
</tr>
<tr>
<td>LS</td>
<td>2</td>
<td>01</td>
<td>Load into a TempRegister (Rout)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>Store from a TempRegister (Rout)</td>
</tr>
<tr>
<td>PointerSel(3:0)</td>
<td>4</td>
<td></td>
<td>Select the pointers from the registers</td>
</tr>
<tr>
<td>MSB_LSB</td>
<td>1</td>
<td>0</td>
<td>Load LSB of the S-word</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Load MSB of the S-word</td>
</tr>
<tr>
<td>XZpointers</td>
<td>1</td>
<td>0</td>
<td>(x_r_pointer = x_p_pointer)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(z_r_pointer = z_p_pointer)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>(x_r_pointer = x_q_pointer)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(z_r_pointer = z_q_pointer)</td>
</tr>
</tbody>
</table>

#### Basic module

<table>
<thead>
<tr>
<th>Name</th>
<th>#bits</th>
<th>value</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA</td>
<td>1</td>
<td>1</td>
<td>There is an input available for the registers</td>
</tr>
<tr>
<td>MAl</td>
<td>2</td>
<td>00</td>
<td>perform an addition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01</td>
<td>perform a multiplication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>perform an inversion</td>
</tr>
<tr>
<td>Ain</td>
<td>2</td>
<td>-</td>
<td>select the first operand from the temporary registers</td>
</tr>
<tr>
<td>Bin</td>
<td>2</td>
<td>-</td>
<td>select the second operand from the temporary registers</td>
</tr>
<tr>
<td>Rout</td>
<td>2</td>
<td>-</td>
<td>select a temporary register for the output</td>
</tr>
<tr>
<td>PassCtr</td>
<td>8</td>
<td>-</td>
<td>count the passes of the multiplication (\rightarrow) selects an input bit for the multiplier</td>
</tr>
<tr>
<td>SBA</td>
<td>1</td>
<td>0</td>
<td>when a shift-back-action have to be performed after the addition (only during an inversion)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>a standard action is performed</td>
</tr>
<tr>
<td>ResetMultReg</td>
<td>1</td>
<td>1</td>
<td>reset the M-bit register in the multiplier</td>
</tr>
<tr>
<td>RDY</td>
<td>1</td>
<td>1</td>
<td>the operation is executed by the basic module</td>
</tr>
</tbody>
</table>

Implementing Cryptographic Solutions in a Modulisr SmartCard core

Copyright © SafeNet. All rights reserved.
### Register module

<table>
<thead>
<tr>
<th>Name</th>
<th>#bits</th>
<th>value</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataOut</td>
<td>32</td>
<td>-</td>
<td>a 32 bit output value</td>
</tr>
<tr>
<td>ScalarPtr</td>
<td>14</td>
<td>-</td>
<td>pointer that points to the RAM-address of the scalar</td>
</tr>
<tr>
<td>NbrOfBits</td>
<td>8</td>
<td>-</td>
<td>the number of bits of the reduction polynomial</td>
</tr>
<tr>
<td>NbrOfBitsSc</td>
<td>8</td>
<td>-</td>
<td>the number of bits of the scalar</td>
</tr>
<tr>
<td>Fpolynomial</td>
<td>M+1</td>
<td>-</td>
<td>the reduction polynomial</td>
</tr>
<tr>
<td>T1</td>
<td>M+1</td>
<td>-</td>
<td>temporary result 1</td>
</tr>
<tr>
<td>T2</td>
<td>M+1</td>
<td>-</td>
<td>temporary result 1</td>
</tr>
<tr>
<td>T3</td>
<td>M+1</td>
<td>-</td>
<td>temporary result 1</td>
</tr>
<tr>
<td>T4</td>
<td>M+1</td>
<td>-</td>
<td>temporary result 1</td>
</tr>
<tr>
<td>SBit</td>
<td>1</td>
<td>-</td>
<td>value of the pointed bit from the s-word</td>
</tr>
</tbody>
</table>

### Execution module

| RoutBus      | M+1   | -     | the result of the performed operation          |

### Interface module

| DatabusIN    | 32    | intern Databus IN                            |
|--------------|-------|-------|------------------------------------------------|
| CPDOUT       | 32    | Databus OUT to CPU/Memory                    |
| PADDR        | 32    | Addressbus to Memory                         |
| WordCountr   | 3     | counts which word will be loaded/stored into/from the Tempregisters |
| RegSel       | 4     | Register Selection for MCR or LDC instructions |
| Sel_MCR_LDC  | 1     | LDC instruction (STC oRW = 1)                 |
|              | 1     | MCR instruction (MRC oRW = 1)                 |
| RW           | 1     | Read(=0) or Write(=1) (internal to registers) |
| BSY          | 1     | Load/Store is ready (intern)                  |
| CDP_Reset    | 1     | the PIP registers has to be reset             |
| Addition     | 1     | a CDP-Addition is asked                      |
| Multiplication | 1     | a CDP-Multiplication is asked                |
| Inversion    | 1     | a CDP-Inversion is asked                     |
| Adding       | 1     | a CDP-Adding is asked                        |
| Doubling     | 1     | a CDP-Doubling is asked                      |
| Scalar       | 1     | a CDP-Scalar is asked                        |
| CMAS         | 2     | coprocessor memory access size               |
| nCPRW        | 1     | coprocessor read/write                       |
| nCPMREQ      | 1     | coprocessor memory request                   |
| CPDRIVEDIN   | 1     | coprocessor drive DIN                        |
| CPDRIVE      | 1     | coprocessor drive                             |
| CPA          | 1     | coprocessor acknowledge                      |
| CPB          | 1     | coprocessor busy                             |
| nCPWAIT      | 1     | coprocessor wait                             |
| CPClkEn      | 1     | coprocessor clock enable                     |
Appendix T. Timing diagrams of the basis operations in polynomial based ECC

T.1 Polynomial Addition

- clock
- MAI
- 0
- MAI0
- Bin
- 0
- Ain
- 0
- SBA normal addition
- SBA shift-back addition
- AIS
- Rout
- 0
- IA
- Bop->T1
- loadB
- Aop->T2
- loadA
- Res->T3
- calcRes
- Result
- T4
- not used
- state in control module
- add
- next state
-
<table>
<thead>
<tr>
<th>State</th>
<th>Instr.</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>pas_md</td>
<td>idle</td>
</tr>
<tr>
<td>first pass</td>
<td>pr_md</td>
<td>pr_md</td>
</tr>
<tr>
<td>b-operand</td>
<td>pr_md</td>
<td>pr_md</td>
</tr>
<tr>
<td>not used</td>
<td>pas_md</td>
<td>idle</td>
</tr>
<tr>
<td>1-bit operand</td>
<td>pr_md</td>
<td>pr_md</td>
</tr>
<tr>
<td>idle</td>
<td>pas_md</td>
<td>idle</td>
</tr>
</tbody>
</table>

Diagram shows 2 Polynomial Multiplication.
T3 Polynomial Inversion

Note: This is an example execution sequence for inputs (W, X) and initial state. (W, X) = (0, 1).
Appendix U. State diagrams final implementation

U.1 The scalar FSM in the control module of the affine case

```
start Scalar and nr_bits_scalar != 0
address = scalar_pointer + (nr_bits_scalar - 1) >> 5
firstpass = true
s_counter = nr_bits_scalar - 1
b = 0

1: s_counter = 0
and b = 0
2: uoad = 0
3: b = 0
FinishedDoubling
b = 0
2: uoad = 0

if (b = 0)
{
b = 1
s_counter--
}

4: b = 1
Not Finished
if (b = 0)
{
s_counter--
}

if (true == firstpass)
{
mask 16 bit of 32-bit s_word:
if ((s_counter >> 4) & 0x1 == 1)
16 LSB of s_reg = 16 LSB s_word
else
16 MSB of s_reg = 16 MSB s_word
firstpass = false
}

5: b = pointed bit by:
(s_counter & 0x1F)
FinishedDoubling

6: read bit b
7: adding
8: reset b
9: wait or load s_word

10: get s-word

Idle

1: nr_bits_scalar = 0
or
nr_bits_scalar = 1
firstcalculation = true

1: load = false
if (s_counter = 0)
{
load
firstcalculation = false
if (b = 0)
{
s_counter--
}
}

firstcalculation = true

start Scalar and nr_bits_scalar != 0
address = scalar_pointer + (nr_bits_scalar - 1) >> 5
firstpass = true
s_counter = nr_bits_scalar - 1
b = 0

1: s_counter = 0
and b = 0
2: uoad = 0
3: b = 0
FinishedDoubling
b = 0
2: uoad = 0

if (b = 0)
{
b = 1
s_counter--
}

4: b = 1
Not Finished
if (b = 0)
{
s_counter--
}

if (true == firstpass)
{
mask 16 bit of 32-bit s_word:
if ((s_counter >> 4) & 0x1 == 1)
16 LSB of s_reg = 16 LSB s_word
else
16 MSB of s_reg = 16 MSB s_word
firstpass = false
}

```
U.2 The scalar FSM of the control module in the projective Montgomery case

```
if b = 0 then
    xr_pointer = xp_pointer
    zr_pointer = zp_pointer
    XZ_pointer = 0
else
    xr_pointer = xq_pointer
    zr_pointer = zq_pointer
    XZ_pointer = 1
fi
```

```
address = scalar_pointer + (s_counter>>5)
l_load = 32
```

```
not FinishDoubling
```

```
not FinishAdding
```

```
\begin{align*}
    b &= (s_word>>((s_counter & 0x1F)) & 0x1 \\
    l_load &= s_counter-- \\
    if b = 0 then \\
    xr_pointer &= xp_pointer \\
    zr_pointer &= zp_pointer \\
    XZ_pointer &= 0 \\
    else \\
    xr_pointer &= xq_pointer \\
    zr_pointer &= zq_pointer \\
    XZ_pointer &= 1 \\
    fi
\end{align*}
```
U.3 Multiplication FSM of the basic module

IA = false
RDY = false
if MAI = '01'
bit_cntr = m-1

Ai_pointer = Abit_cntr
bit_cntr = bit_cntr - 1
resetMultReg = true

Ai_pointer = Abit_cntr
bit_cntr = bit_cntr - 1
resetMultReg = false

IA = true
RDY = true

U.4 Inversion FSM of the basic module

For calculating: \( W = W + X \)
- \( IA = '1' \)
- \( Bin = '00' \)
- \( Rout = '00' \)

RDY = '1'
- \( IA = '0' \)
- \( Bin = '01' \)
- \( Rout = '11' \)

If \( IR = '1' \)
- \( IA = '0' \)
- \( RDY = '1' \)

else if \( W0 = '0' \)
For calculation of:
- \( Y = (Y + Y0*N) >> 1 \)
- \( IA = '0' \)
- \( Bin = '01' \)
- \( Rout = '01' \)

else if \( X0 = '0' \)
For calculation of:
- \( Z = (Z + Z0*N) >> 1 \)
- \( IA = '0' \)
- \( Bin = '00' \)
- \( Rout = '00' \)

else if \( CXW = '0' \)
For calculation of:
- \( Z = Y + Z \)
- \( IA = '1' \)
- \( Bin = '01' \)
- \( Rout = '01' \)

else
For calculation of:
- \( Y = Y + Z \)
- \( IA = '1' \)
- \( Bin = '01' \)
- \( Rout = '01' \)

If \( IR = '1' \)
- \( X1 = '0' \)
- \( W0 = '0' \)
- \( CXW = '0' \)

It is not possible to check the new \( W0 \) immediately, therefore check \( W1 \) instead
- \( IA = '0' \)
- \( Bin = '10' \)
- \( Rout = '00' \)

else if \( XO = '0' \)
For calculation of:
- \( Z = (Z + Z0*N) >> 1 \)
- \( IA = '0' \)
- \( Bin = '00' \)
- \( Rout = '00' \)

else if \( CXW = '0' \)
For calculation of:
- \( Z = Y + Z \)
- \( IA = '1' \)
- \( Bin = '01' \)
- \( Rout = '01' \)

else
For calculation of:
- \( Y = Y + Z \)
- \( IA = '1' \)
- \( Bin = '01' \)
- \( Rout = '01' \)
Appendix V. Global architecture of the register module

32-bit from RAM

pointers + constant values

RegisterSelect(3..0)

32-bit to RAM

MCRorLDC

WordNrRAM(2..0)

RoutBus

LSO

Create selection signals IA LS0 WordNrRAM(2..0)

SigRoute SigSelectj

32^j + 31 < x <= 32^j

SigSelect(0..3)

M-233 M-224 M-223 M-192 M-191 M-166 M-159 M-128 M-127 M-96 M-95 M-64 M-63 M-32 M-31

T4 T3 T2 T1

Appendices

Copyright © SafeNet. All rights reserved.