innovations
for high
performance
microelectronics

Brandenburg
University of Technology
Cottbus - Senftenberg

Master Thesis

SCA Resistent Implementation of the Montgomery *kP*-Algorithm

Estuardo Alpírez Bock
Matriculation nr.: 2818626

24th September 2015

Adviser:	Prof. Dr. rer. nat. Peter Langendörfer
Adviser:	Prof. Dr.-Ing. H. T. Vierhaus
Supervisor:	Dr. Zoya Dyka

Eidesstattliche Erklärung

Der Verfasser erklärt an Eides statt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Ort, Datum Unterschrift des Verfassers

Abstract

Mathematically, cryptographic approaches are secure. This means that the time an attacker needs for finding the secret by brute forcing these approaches is about the time of the existence of our world. Practically, an algorithm implemented in hardware is a device that generates a lot of additional data during the calculation process. Its power consumption, electromagnetic radiation, etc. can be measured, saved and analysed for key extraction. Such attacks are called side channel analysis attacks and are significant threats when applying cryptographic algorithms. By considering these attacks when implementing a cryptographic algorithm, it is possible to design an implementation that is more resistant against them.

The goal of this thesis was to design a methodology to securely implement the Montgomery kP-operation using an IHP implementation as a starting point. In addition, the area and energy consumption of the secure Montgomery kP-multiplier should still be highly efficient. The resistance against power analysis attacks of two different IHP ECC implementations was analysed in this thesis. A horizontal power analysis attack using the *difference-of-means* test was performed with the goal of finding potential leakage sources exploited in side channel analysis attacks, i.e. finding the reasons of a correct extraction of the cryptographic key. For both analysed ECC designs, four key candidates were extracted with a correctness of 90% or more. Through analysis of the implemented Montgomery kP-algorithm's functionality and its power consumption, it was established that the algorithm's operation execution flow was the main cause of the implementations' vulnerability. Thus, a design methodology consisting in changing the Montgomery kP-algorithm operation flow was developed. As a result, the re-designed implementations do not deliver any correctly extracted key candidates whenever the difference-of-means test is performed on them. These re-designs implied an increase on the chip area by about 5% for each implementation. The execution time needed for performing a complete kP-operation was reduced for both designs. Thereby one implementation's execution time was reduced by 12% in comparison to its original version and even though its power consumption was increased by 9%, its energy consumption per kP-operation was reduced by 4.5%.

Kurzfassung

Standardisierte kryptographische Algorithmen sind aus mathematischer Sicht sicher. Dies bedeutet, dass ein Brute-Force-Angriff zur Bestimmung des geheimen Schlüssels einen Zeitaufwand von der Dauer der Existenz unserer Welt hat. In Hardware implementierte Algorithmen generieren aber während des Berechnungsvorganges eine große Menge zusätzlicher Daten. U.a. können der Energieverbrauch des Gerätes sowie seine elektromagnetische Strahlung gemessen, gespeichert und analysiert werden, um den privaten Schlüssel zu extrahieren. Solche Angriffe werden Seitenkanalangriffe genannt und sind erhebliche Bedrohungen für die Sicherheit kryptographischer Algorithmen.

Die vorliegende Arbeit hatte das Ziel, eine Methodik zur Implementierung der Montgomery kP-Operation zu entwickeln, welche Resistenz gegen Seitenkanalangriffe lieferte. Dabei wurde eine IHP Implementierung als Ausgangspunkt benutzt. Zusätzlich sollten die Fläche und der Energieverbrauch der sicheren Montgomery kP-Multiplizierer hoch effizient sein.

Im Rahmen dieser Masterarbeit wurde die Resistenz gegen Seitenkanalangriffe zweier unterschiedlicher IHP ECC Implementierungen analysiert. Ein Power-Analysis-Angriff wurde anhand des *difference-of-means* Testes (DoMT) durchgeführt, um mögliche Sicherheitslücken im Bezug auf Seitenkanalangriffe zu finden, d. h. um die Gründe einer erfolgreichen Schlüssel-Extrahierung festzustellen. Für beide Implementierungen wurden vier Schlüsselkandidaten mit einer Korrektheit von mindestens 90% extrahiert. Nach Analyse der Funktionalität des implementierten Montgomery kP-Algorithmus und seines Momentanleistungsverbrauchs wurde festgestellt, dass die Ausführungseihenfolge der Operationen des Algorithmus die Hauptursache des erfolgreichen Angriffes war. Hierauf aufbauend ist eine neue Methodik zur Implementierung des Montgomery kP-Algorithmus entwickelt worden. Diese Methodik basiert auf einer neuen Ausführungsreihenfolge der einzelnen Operationen im Algorithmus. Nach diesen Änderungen konnten mit dem DoMT keine Schlüssel mehr erfolgreich extrahiert werden. Die Änderungen verursachten eine Erhöhung der Implementierungsflächen um ca. 5%. Die Ausführungszeit einer kompletten kP-Operation ist für beide Implementierungen reduziert worden. Dabei wurde die Ausführungszeit z. B. einer Implementierung im Vergleich zur originalen Version um 12% reduziert und obwohl ihre durchschnittliche Leistung um 9% erhöht wurde, ist ihr Energieverbrauch pro kP-Operation um 4,5% reduziert worden.

Contents

List of Figures ix

List of Tables xi

1 Introduction 1

2 Elliptic Curve Cryptography 5

2.1 Binary-extension field $GF(2^{233})$ . . . 6

2.2 Elliptic Curves over $GF(2^{233})$ . . . 8

2.2.1 Point operations . . . 8

2.3 Elliptic Curve Point Multiplication . . . 9

2.4 Cryptographic Operations . . . 13

2.4.1 Key generation . . . 13

2.4.2 Encryption scheme . . . 14

3 Side-Channel Analysis 17

3.1 Power Analysis . . . 18

3.1.1 Simple Power Analysis . . . 20

3.1.2 Countermeasures against Simple Power Analysis . . . 22

3.1.3 Differential Power Analysis . . . 24

3.1.4 Comparative Power Analysis . . . 26

3.1.5 Countermeasures against Differential and Comparative Power Analysis 29

3.2 Fault Sensitivity Analysis . . . 31

3.2.1 Fault Injection . . . 31

3.2.2 FSA Attack Technique . . . 32

3.2.3 Countermeasures against Fault Sensitivity Analysis . . . 34

4 Vulnerability Assessment of an IHP ECC Design 37

4.1 IHP ECC Design . . . 37

4.2 Difference-of-Means Test . . . 38

4.3 Technical Description . . . 47
4.3.1 System Architecture . . . 47
4.3.2 Control Signal `cntr` Description . . . 50
4.3.3 Montgomery kP-Algorithm Implementation . . . 52
4.4 Observations . . . 58

5 Re-design of the IHP ECC Implementation 71
5.1 Technical Changes . . . 71
5.1.1 System Architecture . . . 72
5.1.2 Control Signal . . . 73
5.1.3 Re-designed Implementation of the Montgomery kP-Algorithm . . . 75
5.2 Observations . . . 91
5.3 Results of the Difference-of-Means Test . . . 94
5.4 Results of a Comparative Power Analysis Attack . . . 100

6 Verification of the Re-design Methodology 103
6.1 Re-design of the ECC Implementation with the 6-clock-cycle Multiplier . . . 103
6.1.1 Re-design of the Main Loop . . . 103
6.1.2 Re-design of the First Loop Iteration . . . 107
6.1.3 Chip Area and Power Consumption of the new Design . . . 108
6.2 Results of the Difference-of-Means Test . . . 108

7 Conclusions 111

Appendix A Acronyms 115

Appendix B Flow Diagrams with Power Traces 117

Appendix C Re-designed Flow Charts 127

Bibliography 135

List of Figures

1.1 Security features and how they can be achieved 2

3.1 Inverter 18
3.2 Power Analysis and different ways of performing it 19
3.3 PT part corresponding to one point doubling and one point addition 21
3.4 PT part corresponding to the execution of the double-and-add algorithm . . . 21
3.5 PT part corresponding to one point doubling and one point addition (unified) 23
3.6 PT part corresponding to an execution sequence of the kP-operation 23
3.7 Vertical and horizontal SCA attacks 25
3.8 Comparative Power Analysis 27
3.9 Collision-based attack 29
3.10 Clock glitch 31
3.11 Fault Sensitivity analysis 33
3.12 Fault Sensitivity comparison 34

4.1 First 8 slots of the simulated PT for case 1 40
4.2 All slots from Figure 4.1 shown clockwise in the same coordinates 40
4.3 Variances as a 57-point-long curve for case 1 41
4.4 Variances as a 57-point-long curve for case 2 42
4.5 Relative correctness curve for the three comparison models 43
4.6 Relative correctness curve using the “direct” and “inverted” assumptions . . . 44
4.7 Relative correctness curves for cases 1 and 2 44
4.8 Relative correctness curves of the ECC blocks 46
4.9 Structure of the IHP ECC design 48
4.10 Configuration of the signal `cntr` 50
4.11 Flow chart diagram for the program `montk1` 54
4.12 Flow chart diagram for the program `montk0` 55
4.13 PT part simulated with the IHP ECC design 59
4.14 Flow chart diagram for `montk0` with PTs simulated during its execution . . . 62

4.15 Flow chart diagram for montk1 with PTs simulated during its execution . . . 63
4.16 Comparison of the power consumptions of all registers 66
4.17 Flow chart of the initialization part of the Montgomery kP-algorithm 68
4.18 Simulated PTs for keys with different values for k_{l-2} 69

5.1 New structure of the IHP ECC design 73
5.2 New configuration of the signal cntr 74
5.3 Flow chart diagram of montk1 re-designed 80
5.4 Flow chart diagram of montk0 re-designed 81
5.5 Flow chart diagram of montk1 in its variant form 82
5.6 Flow chart diagram of montk0 in its variant form 83
5.7 Flow chart diagram of montk0 with corresponding PTs 85
5.8 Flow chart diagram of montk1pre 88
5.9 Flow chart diagram of montk0pre 89
5.10 Flow chart of the re-designed initialization part of the Montgomery kP-algorithm 90
5.11 PT part simulated for the re-designed ECC implementation 91
5.12 PT simulated at the beginning of the execution of the kP-operation 92
5.13 First 8 regular slots of the simulated PT for case 1 95
5.14 Variances as 54-point-long curves for case 1 and case 2 96
5.15 Relative correctness of the key extraction as a curve for case 1 96
5.16 Relative correctness curves for cases 1 and 2 97
5.17 Relative correctness curves of the re-designed ECC blocks 99
5.18 PTs measured from the re-designed ECC implementation executed on an FPGA 101
5.19 Calculated difference of the measured PTs 102

6.1 Flow chart diagram of montk0 adapted to the the 6-clock-cycle multiplier . . 105
6.2 PT part simulated with the re-designed ECC implementation (6-clock) 106
6.3 PT simulated at the beginning of the execution of the kP-operation 107
6.4 Relative correctness curve for the original ECC implementation (6-clock) and for its re-designed version 109

7.1 Relative correctness curves for the original ECC implementation with 9-clock-cycle multiplier and its re-design version 112

List of Tables

1.1 NIST recommended key sizes (in bits) for RSA and ECC 3

3.1 Propagation delays and energy consumptions for an inverter. 18

3.2 Execution sequence of kP and $k(2P)$ with the double-and-add-always algorithm 28

3.3 Sequence of point doublings and point additions for an EC point multiplication 32

4.1 Description of the input values for the Bus. 51

4.2 Values written to the Bus during cycles 56 and 57 (0) 65

5.1 New possible values of signal `cntr` for the Bus 75

5.2 Example of the new execution sequence of `montk1` and `montk0` 77

5.3 Total cell area of the different versions of the ECC implementation 93

5.4 Power, time and energy consumption of the different versions of the ECC implementation 94

6.1 Power, time and energy consumption of the two versions of the ECC Design . 108

7.1 Comparison of the results of the original ECC implementation with the 9-clock-cycle multiplier and its re-designed version 113

7.2 Comparison of the results of the original ECC implementation with the 6-clock-cycle multiplier and its re-designed version 113

1 Introduction

Wireless Sensor Networks (WSNs) are used for monitoring and protecting critical infrastructures as well as in industrial automation. The sensor nodes used in such fields are small sized, low cost devices. Since they are battery powered objects connected to a network, the performance of their functionalities should be efficient over a long period of time (long lived) and the communication between and with these devices should be reliable and secure. The wireless sensor nodes' communication needs to be protected in order not to compromise the functionality of the system they are being used in. This can be achieved by cryptography.

Cryptographic means need to be applied to provide confidentiality, data integrity and availability as security goals for the system [1]. Symmetric and asymmetric cryptographic approaches help reaching these goals, as Figure 1.1 illustrates. This diagram shows how the two types of cryptographic approaches can be applied in order to provide systems such as WSNs with characteristics that define them as secure. For ensuring these features, cryptographic means such as encryption, decryption and digital signature operations have to be applied. Energy efficient methods for implementing such functionalities need to be studied and applied in order to cope with resource constraints of WSNs.

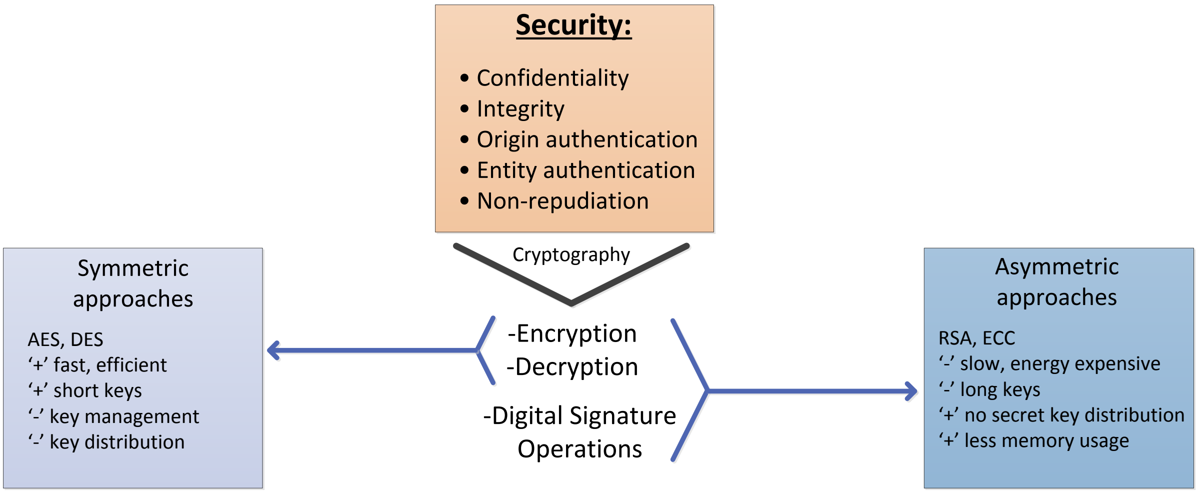


Figure 1.1: Security features and how they can be achieved.
Symmetric and asymmetric cryptographic approaches are compared, pointing out their advantages and disadvantages with '+' and '-' respectively.

In symmetric-key cryptography, a sender and receiver share a single secret key for encryption and decryption of the transmitted message. Algorithms of this type of cryptography are fast, but the key needs to be distributed through a secret channel. In some scenarios, each entity needs to store one key for each communication partner. If a network consists of n sensor nodes, each sensor node stores $(n-1)$ keys[1].
Public-key cryptography is based on complex algorithms that demand more time and energy than those used in symmetric-key cryptography. These algorithms provide numerous benefits for the scenarios they are implemented in. Each party has its own key pair: one public and one private key. The private key has to be stored secretly, while the public key is accessible for everyone who wants to communicate with its owner. Entities in networks with big numbers of communication partners also benefit from this approach, as they do not need to store the private key of every other entity inside the network. This saves storage memory and energy. The non-repudiation problem is also solved with the generation of digital signatures, providing data origin authentication and data integrity.

RSA is the most widespread public-key cryptosystem. Since its development in 1977 [2], RSA gained popularity as it was the first known asymmetric cryptographic approach. It was

[1] In other scenarios, the same secret key is used for all communication partners. This way each node only needs to store this key. These scenarios are not ideal for the security of the network since one successful key extraction would be enough for knowing the value of all communication partners' private key.

included in informative notes from the ISO 9796 [3]. RSA was patented in the United States only [4] and the patent expired in 2000, thus making its application easier.
As an alternative asymmetric cryptography approach, Elliptic Curve Cryptography (ECC) was proposed in 1985 [5] and 1987 [6]. ECC's cryptographic strength is as robust as RSA's, using smaller key-sizes and thus, smaller processing efforts. Table 1.1 shows the key-sizes for both approaches in order to provide an equivalent security level.

Table 1.1: NIST recommended key sizes (in bits) for RSA and ECC (binary field) [7], [8].

RSA Key Size	**ECC Key Size**
1024	163
2048	233
3072	283
7680	409
15360	571

Mobile sensor nodes can benefit from operating with small key-sizes. This decreases the amount of data needed to be transmitted, stored and processed when communicating, which reduces their memory usage, power consumption, and time of communication. In consideration of wireless sensor nodes and other devices with limited energy resources, ECC is the preferred crypto-system to provide secure communication between these devices. Nevertheless, ECC has gone through a slow adoption. One of the reasons for this is patent related. Different aspects of ECC have been patented by different persons and companies around the world. The Canadian company Certicom [9] itself holds more than 130 patents related to elliptic curves and asymmetric cryptography in general [7]. This makes its application for commercial purposes more difficult. Despite these difficulties, there is a lot of research done in the field of ECC because of the previously mentioned benefits.

Though mathematically secure, the implementation and execution of cryptographic algorithms in hardware and software lead to measurable physical parameters, such as power consumption, electromagnetic radiation and execution time of the operations performed. If a device such as a wireless sensor node ends up in the hands of an attacker, the attacker would be able to measure and analyse these parameters. The information provided by the measurements can help him reaching his goal: determining the value of the private key. Such attacks are known as Side-Channel Analysis (SCA) attacks. These aspects have to be taken into account when implementing a cryptographic algorithm.

For the work described in this thesis a vulnerability assessment of an ECC implementation from Innovations for High Performance Microelectronics (IHP) [10] was performed. This implementation had proven to be vulnerable against SCA attacks. A private key could be successfully extracted by performing a *difference-of-means* test based on the power consumption of the design. The assessment was made in order to find out the reasons that made this implementation vulnerable to the difference-of-means test. After performing the assessment, the IHP ECC implementation was re-designed in order to make it more robust against SCA.

Several changes were implemented and their effectiveness was proved by performing a power analysis attack using the difference-of-means test again.

The re-design method proposed in this thesis was applied for two ECC designs: the first design — the IHP hardware accelerator for ECC with a 9-clock-cycle multiplier[2] — was analysed in order to understand the parts of it that needed to be re-designed. The same re-design ideas were then applied to the IHP hardware accelerator for ECC with a 6-clock-cycle multiplier[3] in order to evaluate the applicability of the re-design method to other implementations.

The rest of this thesis is structured as follows: Chapter 2 introduces the elliptic curve crypto-systems and their mathematical background. Chapter 3 provides an introduction to SCA. This description is focused on Power Analysis (PA) attacks, as these are some of the most studied SCA attacks. This includes a description of the horizontal PA attack using the difference-of-means test. This attack was performed on the IHP ECC implementations. Results are described in Chapter 4. Chapter 4 also describes the vulnerability assessment performed on the IHP ECC design. This description was made based on analysis of the source code and simulations of the functionality and power consumption of the design. The procedure for re-designing the ECC implementation is described in Chapter 5, pointing out how it has been managed to improve its resistance against the described SCA attacks and to optimize its energy consumption. Chapter 6 describes how this re-design ideas can be applied to other versions of the ECC design using the implementation with the 6-clock-cycle multiplier as an example. Conclusions are given in Chapter 7.

[2] The multiplier entity that forms part of this design requires 9 clock cycles for completing one multiplication.
[3] The multiplier entity that forms part of this design requires 6 clock cycles for completing one multiplication.

2 Elliptic Curve Cryptography

The concept of using elliptic curves for designing asymmetric cryptographic systems was first proposed in 1985 [5]. ECC is based on elliptic curves defined over finite fields, also known as Galois Fields (GF). These are fields that consist of a finite number of elements, denoted as their *order*, and a set of two operations: addition and multiplication. The commutative and distributive laws are valid for operations performed between the elements of a field. Finite fields are finite cyclic groups, this means that any mathematical operation between two elements of the field will result in another element of the same field.

Standard elliptic curves used for cryptographic implementations can be defined over fields of a prime basis, denoted as $GF(p)$ where the order p is a prime number, or over binary-extension fields, denoted as $GF(2^m)$ where m is a positive integer. Elements in $GF(2^m)$ can be represented as up to m-bit long binary numbers. While prime-order curves are well suited for software implementations, elliptic curves defined over $GF(2^m)$ can be efficiently implemented in hardware [11]. For this reason, the IHP hardware accelerator for ECC, analysed and re-designed in this work, was implemented for the elliptic curve B-233, defined over $GF(2^{233})$. This finite field was selected since key sizes of 233 bits provide cryptographic strength at least until the year 2030 [8].

This chapter gives a brief introduction to ECC, describing its basic mathematical background. The definition of the elements on $GF(2^{233})$ and the operations performed with them are given. The operations with elliptic curve points are introduced as well. These point operations — the point addition and point doubling — are used to perform the elliptic curve point multiplication, denoted as kP-operation, which is also described in this chapter. Finally, it is explained how these bases are used for approaching key generation and encryption schemes. More detailed information about finite field and elliptic curve arithmetic can be found in sections 2 and 3 of [12].

2.1 Binary-extension field $GF(2^{233})$

The elements of a binary extension field can be represented using a *polynomial basis*. To construct the field, an *irreducible polynomial* is needed. The irreducible polynomial $r(t)$ is a polynomial of degree m, which cannot be factored as a product of polynomials of a degree less than m [12]. Thus, an irreducible polynomial $r(t)$ of degree 233 with coefficients in $GF(2)$[1] can be used to construct the extension field $GF(2^{233})$.

The irreducible polynomial for the elliptic curve B-233 over $GF(2^{233})$ is defined as follows [8]:

$$r(t) = t^{233} + t^{74} + 1. \tag{2.1}$$

The elements of $GF(2^m)$ with an irreducible polynomial $r(t)$ of degree m are then polynomials of degree $m - 1$ with coefficients in $GF(2)$:

$$A(t) = \{a_{m-1}t^{m-1} + a_{m-2}t^{m-2} + \ldots + a_2t^2 + a_1t + a_0 : a_i \in GF(2)\}.$$

The elements of $GF(2^{233})$ are then polynomials with a maximum degree of 232 and their coefficients are either 0s or 1s. These elements can be represented as binary numbers with up to 233 bits: $(a_{232}, a_{231}, ..., a_2, a_1, a_0)_2$.

The irreducible polynomial (2.1) is necessary for performing the multiplication and division of two elements of $GF(2^{233})$ as it will be explained in this chapter.

Operations for addition, subtraction, multiplication and division can be performed between the elements of the field. The following statements are obtained from the definition of these mathematical operations [12] applied to the elements of $GF(2^{233})$.

[1] $GF(2)$ is the smallest finite field. It consists of only two elements: 0 and 1. The addition and subtraction of these two elements is performed by a boolean XOR operation. The multiplication of the two elements is performed with a boolean AND operation.

Addition and Subtraction: Additions and subtractions are performed as bitwise XOR operation of the binary representation of two elements of $GF(2^{233})$. The XOR operator is denoted in this work by $\oplus$.

If $A(t), B(t) \in GF(2^{233})$, an addition (or subtraction) of these two elements is performed in the following way:

$$\begin{aligned} A(t) \pm B(t) &= (a_{232}, a_{231}, \ldots, a_2, a_1, a_0)_2 \oplus (b_{232}, b_{231}, \ldots, b_2, b_1, b_0)_2 \\ &= (a_{232} \oplus b_{232}, a_{231} \oplus b_{231}, \ldots a_2 \oplus b_2, a_1 \oplus b_1, a_0 \oplus b_0)_2 \\ &= A(t) \oplus B(t). \end{aligned}$$

Multiplication: This operation between two elements of $GF(2^{233})$ is performed in two steps. The first step is the multiplication of the polynomials. The product of two polynomials of degree $m-1$ results in a polynomial of degree $2m-2$. For this reason, a reduction step is required after the polynomial multiplication to find the corresponding polynomial of degree $m-1$ in $GF(2^m)$. This is performed by dividing the polynomial product by the irreducible polynomial $r(t)$. The reminder of this division is the corresponding polynomial of degree $m-1$.

If $A(t), B(t) \in GF(2^{233})$, their multiplication is performed in the following way:

$$A(t) \cdot B(t) = (A(t) \cdot B(t)) \bmod r(t).$$

Division: the division in $GF(2^{233})$ is performed by multiplying the dividend by the divisor's inverse. For the IHP ECC implementation this was realized by applying the Fermat's Little Theorem [13]. With this approach, a division is performed as a sequence of square operations and multiplications.

Hardware implementations benefit from the simplicity of the $GF(2^{233})$ field operations: no carries are generated. Thus, the complete hardware implementation of these field operations requires smaller area than the implementation of operations in $GF(p)$.

2.2 Elliptic Curves over $GF(2^{233})$

This section provides the definition of elliptic curves and a short introduction to elliptic curve point operations. It is important to point out that the definitions and descriptions in this section are simplified by precisely characterizing elliptic curves over $GF(2^{233})$ with the irreducible polynomial $r(t)$ (2.1).

An elliptic curve E over $GF(2^{233})$ is defined by the following equation

$$E : y^2 + xy = x^3 + ax^2 + b, \tag{2.2}$$

where a and b are elements of $GF(2^{233})$. A pair of numbers $x, y \in GF(2^{233})$ is called a *point* $\boldsymbol{P}$ on the curve E, if the pair (x, y) satisfies the equation 2.2 [12]. (x, y) are then point coordinates of the elliptic curve.
The elliptic curve B-233 over $GF(2^{233})$ with the irreducible polynomial $r(t)$ (2.1) is defined by equation 2.2, where the coefficients a and b are the following hexadecimal numbers [8]:

$a = 1$

$b =$ 066 647ede6c 332c7f8c 0923bb58 213b333b 20e9ce42 81fe115f 7d8f90ad.

All points $P_i = (x_i, y_i)$, satisfying equation 2.2 and a *point at infinity* $\boldsymbol{O}= (x, \infty)$ are the set of points denoted by $E(GF(2^{233}))$.

The following subsection describes the operations with points in $E(GF(2^{233}))$.

2.2.1 Point operations

This subsection describes the point addition and point doubling operations. This description is made using concrete formulas for elliptic curves over $GF(2^{233})$, based on equation 2.2.

Point addition: given two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$, whereby $P \neq Q$ (i.e. $x_P \neq x_Q$ and $y_P \neq y_Q$), the addition of both determines point $R = (x_R, y_R)$ on the curve E: $R = P + Q$.
The coordinates x_R and y_R are calculated separately with the following formulae:

$$\begin{aligned} x_R &= \lambda^2 + \lambda + x_P + x_Q + a, \\ y_R &= y_P + y_Q + \lambda(x_P + x_Q), \end{aligned} \tag{2.3}$$

with

$$\lambda = \frac{y_Q + y_P}{x_Q + x_P}.$$

Here, a is the coefficient from equation 2.2.

Point doubling: given one point $P = (x_P, y_P)$, the addition of this point with itself determines point $R = (x_R, y_R)$: $R = P + P = 2P$.

For the point doubling operation, the coordinates x_R and y_R are:

$$\begin{aligned} x_R &= \lambda^2 + \lambda + a, \\ y_R &= y_P + y_Q + \lambda(x_P + x_Q), \end{aligned} \tag{2.4}$$

with

$$\lambda = x_P + \frac{y_P}{x_P}.$$

Also here a is the coefficient from equation 2.2.

All mathematical operations performed in formulae 2.3 and 2.4 correspond to the field operations described in the previous section. The calculation of the coefficient λ is complex since it requires a field division. To avoid performing divisions and obtain a faster calculation of the point operations, projective coordinates can be applied [14].
The set of elliptic curve points $E(GF(2^{233}))$ with operations 2.3 and 2.4 form an additive finite Abelian group [12].

The elliptic curve point addition and point doubling operations are necessary to perform the elliptic curve point multiplication, denoted as kP. The kP-operation is described in the following section.

2.3 Elliptic Curve Point Multiplication

Cryptographic operations such as encryption, decryption and key generation are composed of different mathematical operations. The most complex operation of them is the elliptic curve point multiplication kP. It is the most time and energy consuming operation when encryption or decryption is performed. This is why this operation is very often optimized.

An encryption requires two kP-operations while a decryption requires only one. Calculating kP takes more than 90% of the decryption time.

The multiplication of an elliptic curve point P with a scalar k, i.e. the kP-operation, is defined as k times the addition of P to itself:

$$kP = \underbrace{P + P + ... + P}_{k}.$$

The three algorithms presented in this section illustrate the idea behind efficient and *balanced* techniques to protect cryptographic implementations against SCA attacks. In this context, *balanced* means that each calculation of the kP-operation has to be performed by the same type and amount of operations for each bit of the number k, independent of the bit's value.

For algorithm implementation purposes, the number k can be represented as the sum of powers of two, where k is an l-bit long binary number:

$$k = \sum_{i=0}^{l-1} k_i 2^i.$$

The kP-operation can then be represented as follows:

$$kP = \sum_{i=0}^{l-1} k_i(2^i)P = k_0P + k_1(2)P + k_2(2^2)P + \ldots + k_{l-1}(2^{l-1})P. \tag{2.5}$$

Based on this representation of the kP-operation, the following algorithms were designed.

Double-and-add algorithm (left-to-right)

The left-to-right double-and-add algorithm is an implementation of the kP-operation represented in formula 2.5 under the following observation[2]:

$$\begin{aligned}\sum_{i=0}^{l-1} k_i(2^i)P &= k_0P + k_1 2P + k_2 2^2 P + \ldots + k_{l-2}2^{l-2}P + k_{l-1}2^{l-1}P \\ &= k_0P + 2(k_1P + 2(k_2P + \ldots + 2(k_{l-2}P + k_{l-1}2P)\ldots)).\end{aligned}$$

The bits of the scalar k are processed from left to right, i.e. from the Most Significant Bit (MSB) to the Least Significant Bit (LSB). For every bit $k_i = 0$, only point doubling

[2] A direct implementation of the formula 2.5 is the right-to-left double-and-add algorithm.

is performed. For every bit $k_i = 1$, point doubling and point addition are performed as Algorithm 1 shows.

Algorithm 1 Double-and-add algorithm for the kP-operation (left-to-right)

```
Input: $k = (k_{l-1}, ..., k_1, k_0)_2, P \in E(GF(2^m))$.
Output: $kP$.
  $Q \leftarrow O$.
  for $i$ from $l-1$ downto 0 do
    $Q \leftarrow 2Q$.
    If $k_i = 1$ then $Q \leftarrow Q + P$.
  end for
  return $Q$.
```

Since a different number of operations is performed depending on the value of k_i, notable key-dependant patterns would be shown in the power traces measured from a hardware device performing this algorithm [15].

Double-and-add-always algorithm

The double-and-add-always algorithm [16] is more robust against SCA attacks than the straight forward double-and-add. This method ensures that the type and number of elliptic curve point operations performed for each bit k_i of the scalar k are always the same, independent of the value of k_i.

Algorithm 2 Double-and-add-always algorithm for the kP-operation (left-to-right)

```
Input: $k = (k_{l-1}, ..., k_1, k_0)_2, P \in E(GF(2^m))$.
Output: $kP$.
  $Q[0] \leftarrow O$.
  for $i$ from $l-1$ downto 0 do
    $Q[0] \leftarrow 2Q$, $Q[1] \leftarrow Q[0] + P$.
    $Q[0] \leftarrow Q[k_i]$.
  end for
  return $Q[0]$.
```

In Algorithm 2 for each bit k_i a point doubling is performed and saved as point $Q[0]$. Next, a point addition of this saved point $Q[0]$ with the input point P is performed and its result is

saved as point $Q[1]$. Then, the value of k_i is checked: if its value is '0', then $Q[0]$ is processed, otherwise $Q[1]$.
The results of some operations are not used in many iterations, but nevertheless these operations are performed to ensure balance. An increased energy and time consumption is compelled by performing these operations, which is a disadvantage of applying this method.

Montgomery Algorithm

In [17] Peter Montgomery introduced a balanced method for performing the kP-operation, shown in Algorithm 3.

Algorithm 3 Montgomery algorithm for the kP-operation

```
Input: $k = (k_{l-1}, ..., k_1, k_0)_2$ with $k_{l-1} = 1, P = (x, y) \in E(GF(2^m))$.
Output: $kP$.
  $Q[0] \leftarrow P$, $Q[1] \leftarrow 2P$.
  for $i$ from $l - 2$ downto 0 do
    if $k_i = 1$ then
      $Q[0] \leftarrow Q[0] + Q[1]$, $Q[1] \leftarrow 2Q[1]$.
    else
      $Q[1] \leftarrow Q[0] + Q[1]$, $Q[0] \leftarrow 2Q[0]$.
    end if
  end for
  return $Q[0]$.
```

Implementing the Montgomery kP-algorithm in hardware increases the resistance against SCA attacks. Each key bit is processed by the same type, amount and sequence of operations. Julio López and Ricardo Dahab showed in [18] how this algorithm could be improved by using projective coordinates (see Algorithm 4). Only the value of the x-coordinate of the point P is used. No division operations and no operations with y-coordinates of the points need to be performed. For this reason, this algorithm was implemented in the ECC implementation whose analysis and re-design are described in this thesis.

Algorithm 4 Montgomery algorithm for the kP-operation using projective coordinates

```
Input: $k = (k_{l-1}, ..., k_1, k_0)_2$ with $k_{l-1} = 1, P = (x, y) \in E(GF(2^m))$.
Output: $kP = (x_1, y_1)$.
  $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$.
  for $i$ from $l - 2$ downto 0 do
    if $k_i = 1$ then
      $T \leftarrow Z_1$, $Z_1 \leftarrow (X_1Z_2 + X_2Z_1)^2$, $X_1 \leftarrow xZ_1 + X_1X_2TZ_2$,
      $T \leftarrow X_2$, $X_2 \leftarrow X_2^4 + bZ_2^4$, $Z_2 \leftarrow T^2Z_2^2$.
    else
      $T \leftarrow Z_2$, $Z_2 \leftarrow (X_2Z_1 + X_1Z_2)^2$, $X_2 \leftarrow xZ_2 + X_1X_2TZ_1$,
      $T \leftarrow X_1$, $X_1 \leftarrow X_1^4 + bZ_1^4$, $Z_1 \leftarrow T^2Z_1^2$.
    end if
  end for$x_1 \leftarrow X_1/Z_1$.
  $y_1 \leftarrow y + (x + x_1)[X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)]/(xZ_1Z_2)$.
  return $((x_1, y_1))$.
```

2.4 Cryptographic Operations

By understanding the mathematical background of elliptic curves and how their point operations are performed, it can now be described how they are applied for cryptographic means. This section describes how the elliptic curve point operations are used for performing key generation, encryption and decryption operations.

2.4.1 Key generation

A base point G with the following coordinates is selected to generate a key pair to be used when applying curves B-233 [8]:

$$x_G = \text{0fa c9dfcbac 8313bb21 39f1bb75 5fef65bc 391f8b36 f8f8eb73 71fd558b}$$

$$y_G = \text{100 6a08a419 03350678 e58528be bf8a0bef f867a7ca 36716f7e 01f81052}$$

The private key k is an integer chosen randomly from the interval [1,n−1], where n is the order of the base point G. The corresponding public key Q can be computed as: $Q = kG$.

Algorithm 5 Elliptic curve key generation

```
Input: Elliptic curve domain parameters $(p, E, G, n)$.
Output: Public key $Q$ and private key $k$.
  Select $k \in [1, n-1]$.
  Compute $Q = kG$.
  return $(Q, k)$.
```

Although the base point G and the key Q are public and can be known by everybody, it is still difficult to determine the value of k. This is known as the Elliptic Curve Discrete Logarithm Problem (ECDLP).

2.4.2 Encryption scheme

The encryption of a plaintext m can be done in the following way: The plaintext m is represented as the x-coordinate of a point $M = (m, y_M)$. The encryption should be performed by adding M to dQ, where Q is the public key of the receiver and d is a random number. The result of this operation is saved as point C_2. The sender then transmits two elliptic curve points: $C_1 = dG$ and C_2.

Algorithm 6 Elliptic curve encryption

```
Input: Elliptic curve B-233 domain parameters [8], public key of receiver $Q$, plaintext $m$.
Output: Ciphertext: Points $C_1$ and $C_2$.
  Represent $m$ as the $x$-coordinate of point $M \in$ B-233.
  Randomly select $d \in [1, n-1]$.
  Compute $C_1 = dG$.
  Compute $C_2 = M + dQ$.
  return $C_1, C_2$.
```

The receiver uses its own private key k for the decryption:

$$M = C_2 - kC_1.$$

Hereby, $kC_1 = k(dG) = d(kG) = dQ$ and M is obtained by calculating $M = C_2 - dQ$. It is now possible to recover the message m, which is the x-coordinate of point M. Only the owner of the private key k is able to recover M without knowledge of the number d (d is known to the sender but was not sent). Algorithm 7 shows the decryption process.

Algorithm 7 Elliptic curve decryption

```
Input: Elliptic curve B-233 domain parameters [8], private key $k$, ciphertext $(C_1, C_2)$.
Output: Plaintext $m$
  Compute $M = C_2 - kC_1$.
  return $m = x$-coordinate of $M$.
```

From a crypto-analysis point of view, ECC is considered to be secure due to the ECDLP and the proposed key length. Nevertheless, the situation changes completely if SCA attacks are taken into account.

3 Side-Channel Analysis

Devices executing cryptographic algorithms consume energy and generate electromagnetic and infrared radiations amongst others. These physical parameters can provide information about the chip's functionality and data being processed.
Cryptographic hardware implementations are made on Integrated Circuits (ICs) based on Complementary Metal-Oxide-Semiconductor (CMOS)-technology. CMOS transistor gates consume power and time when performing cryptographic tasks. These consumptions are dependant on the input data being processed using the private key and thus, the transistors' activities are principally responsible for information leakage. The current conducted by all transistors of the chip can be measured and analysed with the goal of extracting the private key.
Much research has been done on the different physical parameters that can be observed and used to understand the internal activities of cryptographic devices. As a consequence, countermeasures to make these physical phenomena less observable have been developed. The idea is to make cryptographic devices less vulnerable against physical attacks and in order to achieve that, the implementation of algorithms in hardware needs to be done under consideration of these observable parameters.

Side-Channel Analysis (SCA) attacks are *passive* physical attacks. They are based on the observation of the devices' behaviour when performing cryptographic operations. In contrast to this, *active* attacks aim at manipulating the behaviour of the device under attack.

This chapter describes SCA and corresponding countermeasures for ECC implementations. It focuses on Power Analysis (PA). The work presented in this thesis consists of re-designing a cryptographic implementation in order to improve its resistance against PA attacks. For this reason, several types of PA attacks are described in this chapter. As an example of active attacks based on PA, the Fault Sensitivity Analysis attack is presented. The technique presented in [19], where the initial data execution is used as a template for the extraction of the private key, is described since a countermeasure against this attack was also implemented in this work.

3.1 Power Analysis

CMOS gates are used for implementing logic functions in hardware. The complete power consumption of an IC is composed of the sum of the power consumption of all its gates. The power consumption of a gate is dependable on the technology it is produced in, its type and its inputs. This fact will be explained on example of an inverter from the IHP 130 nm technology [10]. Figure 3.1 shows the representation of an inverter as a gate.

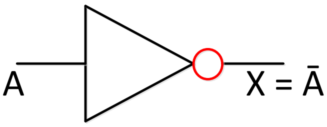


Figure 3.1: Inverter with its input signal A and output signal X.

A change of the input signal A from ‘0’ to ‘1’ (and thus of the output signal X from ‘1’ to ‘0’) causes different propagation delays and energy consumptions compared to a change of A from ‘1’ to ‘0’ (of X from ‘0’ to ‘1’). Table 3.1 shows the propagation delays and energy consumption for both cases as an example.

Table 3.1: Propagation delays and energy consumptions for an inverter.

A	**X**	**delay**	**energy consumption**
$0 \longrightarrow 1$	$1 \longrightarrow 0$	0.02807 ns	0.00265 pJ
$1 \longrightarrow 0$	$0 \longrightarrow 1$	0.02313 ns	0.00239 pJ

A change of A from ‘0’ to ‘1’ causes about 21% longer propagation delays and about 11% higher energy consumptions as a change from ‘1’ to ‘0’. This relation between the propagation delays or energy consumption of gates in a cryptographic chip and the data it processes is the feature that an attacker can exploit for the extraction of the private key.

A circuit consists of many gates. Depending on the instructions performed by the circuit at a certain clock cycle, only some of those gates switch. In the next clock cycle, other gates are switching. The current flow on each clock cycle depends on these switching gates. The power consumption of a device is proportional to its current flow[1]. A device’s power consumption changes in each clock cycle since the device performs different instructions. The purpose

[1] $p(t) = i(t) \cdot u(t)$.

of examining a Power Trace (PT) is to find its relation to the data the device is processing during each clock cycle.

Cryptographic implementations perform operations that depend on the value of their private key. Analysing these operations can help extracting this value. If the type and/or sequence of the processing steps are dependant of the value of the processed key bit, these differences are observable (or easy identifiable) in the PT and make the analysis easier. Such implementations will be referred to as *unbalanced* in this work. If the key-dependent operations do not differ excessively from each other, the variations along a PT's curve would be more difficult to identify.

Three different kinds of PA are described in this section:

- Simple Power Analysis (SPA): the key can be easily extracted through direct observations of the measured PTs without statistical means.
- Differential Power Analysis (DPA): the key can be extracted by applying statistical means.
- Comparative Power Analysis (ComPA): the key can be extracted by comparing a PT measured on a device using a known key and a PT measured on a device using an unknown key.

These different kinds of PA are illustrated in Figure 3.2.

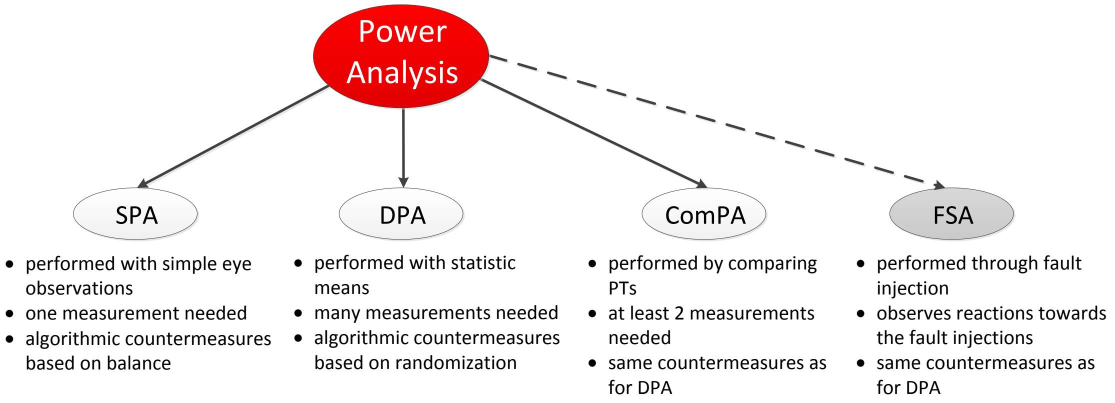

Power Analysis
SPA
DPA
ComPA
FSA
performed with simple eye observations
one measurement needed
algorithmic countermeasures based on balance
performed with statistic means
many measurements needed
algorithmic countermeasures based on randomization
performed by comparing PTs
at least 2 measurements needed
same countermeasures as for DPA
performed through fault injection
observes reactions towards the fault injections
same countermeasures as for DPA

Figure 3.2: Power Analysis and different ways of performing it.

These PA methods are detailed described in the rest of this section, since the work described in this thesis had the goal of re-designing a cryptographic implementation to protect it against PA. Published countermeasures against these attacks are described as well. Additionally, the Fault Sensitivity Analysis (FSA) is shown in Figure 3.2. FSA attacks are not classified as SCA attacks since they involve the direct manipulation of the analysed implementations. For this reason, FSA is described at the end of this chapter.

3.1.1 Simple Power Analysis

SPA is a technique that uses only one measured PT for extracting the key [15]. For ECC, a PT of the kP-operation (see section 2.3) is analysed. Unbalanced ECC implementations can be successfully analysed in short time. If different types or a different number of operations is performed depending on the value of the processed key bit, differences are observable in the corresponding parts of the PTs. The complete PT can then be divided into *slots*[2]. The slots differ in this case depending on the actual value of k_i.
For example, an implementation based on the double-and-add algorithm (see Algorithm 1 on p. 11) would cause a PT of this type, having two different types of slots. This algorithm performs one point doubling operation for each iteration of its main loop for the case $k_i = 0$. For the case $k_i = 1$, one point doubling and one point addition are performed. In this case, the slots for the case $k_i = 0$ are shorter than those for $k_i = 1$. It is thus easy to determine for which case k_i the loop iteration has been performed.

The next helpful fact for analysing measured PTs of Elliptic Curve (EC) kP-operations is that the point operations, the doubling and addition, are performed using different formulas each (see section 2.2.1). A point doubling consists of less arithmetic operations than a point addition. Thus, the addition is longer than the doubling and so, the part of a PT that corresponds to an addition is longer than the part corresponding to a doubling. Figure 3.3 shows a part of a PT with one point doubling and one point addition; they can be clearly distinguished.

[2] A slot is a part of a PT which corresponds to the processing of one key bit.

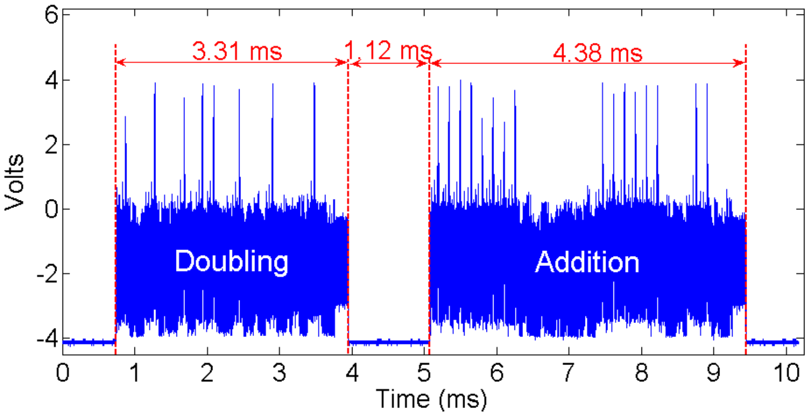


Figure 3.3: Part of a PT of an EC kP-operation showing one point doubling and one point addition (source: [20]).
The point doubling is considerably shorter than the point addition.

ECC implementations with distinguishable EC point operations and/or with algorithmic steps dependable on the key bit value are vulnerable to SPA attacks. By observing the PT of one complete execution of such algorithms, the individual bits of the private key can be extracted. Figure 3.4 shows an example of such PTs.

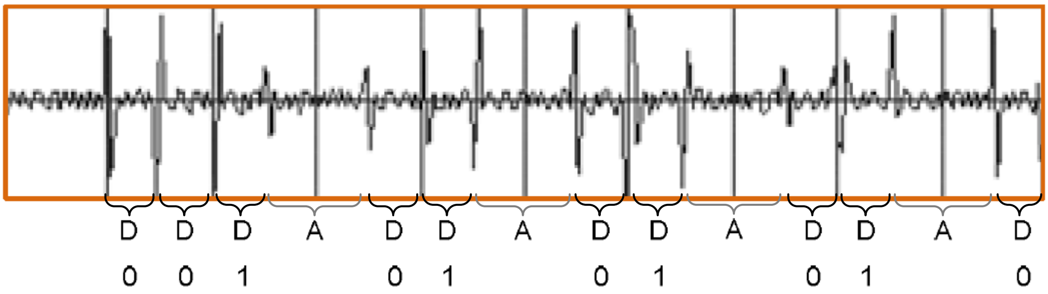


Figure 3.4: Part of a PT of an EC kP-operation implemented using the double-and-add algorithm (source: [21]).
The point doubling **D** is considerably shorter than the point addition **A**. By recognizing the execution of a point doubling followed by a point addition, it is known that a key bit $k_i = 1$ has been processed. Otherwise, a key bit $k_i = 0$ has been processed.

3.1.2 Countermeasures against Simple Power Analysis

The basic idea for making a cryptographic implementation robust against SPA attacks is to make it non-dependable on the key bit value, i.e. to perform always the same number of the same operations and in the same order. Two possibilities for achieving this are listed below.

Double-and-add-always algorithm. In [16] a method for protecting implementations of the double-and-add algorithm was proposed: the point addition would always be executed after the point doubling, but its result would be ignored in some cases (*dummy* operations). The loops of the double-and-add-always algorithm (see Algorithm 2 on p. 11) do not excessively differ from each other. The same point operations, i.e. one point addition and one point doubling, are always performed, independently of the key bit value. Thus the extraction of the private key becomes more difficult.

As mentioned in section 2.3, a disadvantage of algorithms that use dummy operations is the unnecessary energy and time consumed by performing more computations as it is required.

Indistinguishable EC Point Operations. Alternatively, the formulae for the point addition and point doubling (see section 2.2.1) can be *unified* [22]. In this case, both point operations consist of the same arithmetic operations, which are performed in the same sequence. This is achieved with help of dummy field arithmetic operations.

By using unified formulae for point operations, even if different point operations are performed depending on the value of the key bits, the parts of the PT showing the point additions are indistinguishable from the parts of the PT showing the point doublings. Figure 3.5 shows two parts of a PT with one point doubling and one point addition, which are performed using a unified formula. The length and shape of the traces of these operations look identical to the simple eye. An implementation of the kP-operation using the double-and-add algorithm with unified formulae for point operations would also be protected against SPA.

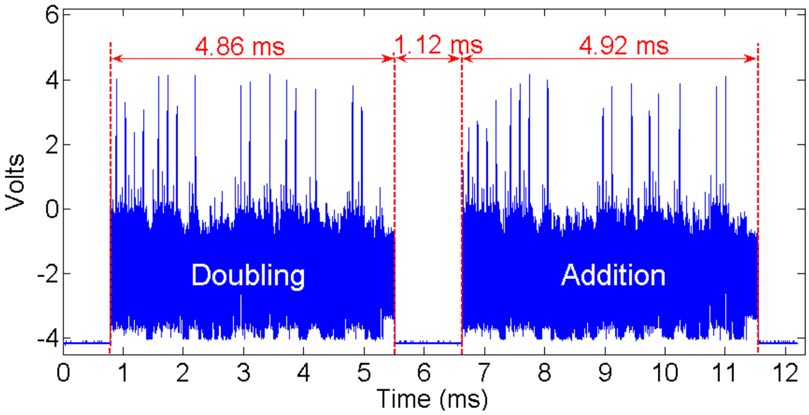


Figure 3.5: Part of a PT with one point doubling and one point addition (source: [20]).
Both EC point operations are performed using a unified formula. Their traces are indistinguishable.

Alternatively, the point addition formula could be implemented in such way, that its trace looks as if two doubling operations are performed [12]. Figure 3.6 shows such an example.

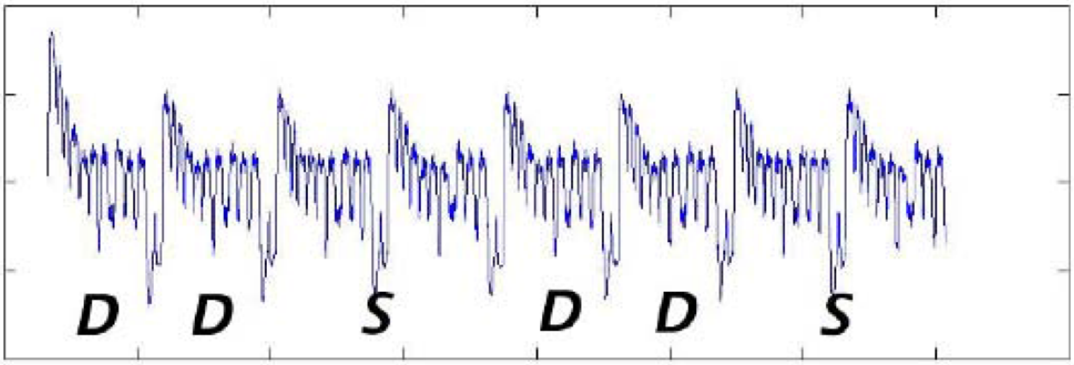


Figure 3.6: Part of a PT of a kP-operation (source: [12]).
The point doubling and point addition are identified by the letters **D** (doubling) and **S** (sum) respectively. The trace of one point addition looks like the trace of two point doubling operations. It is not easy to identify differences between the slots.

For providing protection against SPA, both countermeasures — the double-and-add-always algorithm and a unified formula for EC point operations — can be combined. A disadvantage of implementing the kP-operation this way is that a big number of dummy operations need to be executed. This increases the execution time and energy consumption significantly.

The countermeasures described in this section can provide protection against SPA, but some steps of the double-and-add-always algorithm are still dependable on the value of the key bit. For example, when registers are overwritten. This may not have a visible effect on the PT when observed with the simple eye, but it can be detected using statistical methods. “Small” key bit value dependable differences in each slot of a PT can be used for a successful extraction of the key. Attacks based on this principal are described in the following subsection.

3.1.3 Differential Power Analysis

PTs measured on designs implemented with countermeasures against SPA consist of slots that look similar to each other. Still, “small” key bit dependabilities can be exploited for successfully performing attacks on these designs. For example when conditional instructions are executed, the results of some operations are written in different registers or during different clock cycles depending on the key bit value being processed. This may lead to *light* unbalanced power consumptions.
Considering this fact, the power consumption of a cryptographic device can still leak information about the private key. DPA attacks apply statistical methods on a big number of PTs (or slots of one PT) to find the dependability of the power consumption on the processed data. The noise can be reduced and minor differences between the traces can be uncovered.

A DPA attack can be performed *vertically*, or *horizontally* [23]. Vertical SCA attacks are attacks that need a numerous amount of measurements in order to extract the secret. For each measurement, the implementation performs the same operation with the same private key, but different input parameters. Consequently, a big number of PTs is collected. A certain part in each PT is selected for analysis. This part corresponds to the execution of the same step in the algorithm. Statistical methods are applied to compare all PTs in relation to their selected part. This way, the traces can be grouped according to the differences found and the value of the key bit processed in this part of the execution can be suggested. A horizontal SCA attack can be performed using only one or a few measurements. A PT is divided into slots, each corresponding to an iteration of the implemented algorithm. Since each iteration is performed differently according to the processed key bit value, statistical methods can be applied to analyse the slots the same way the PTs are analysed in vertical attacks. These two concepts are illustrated in Figure 3.7.

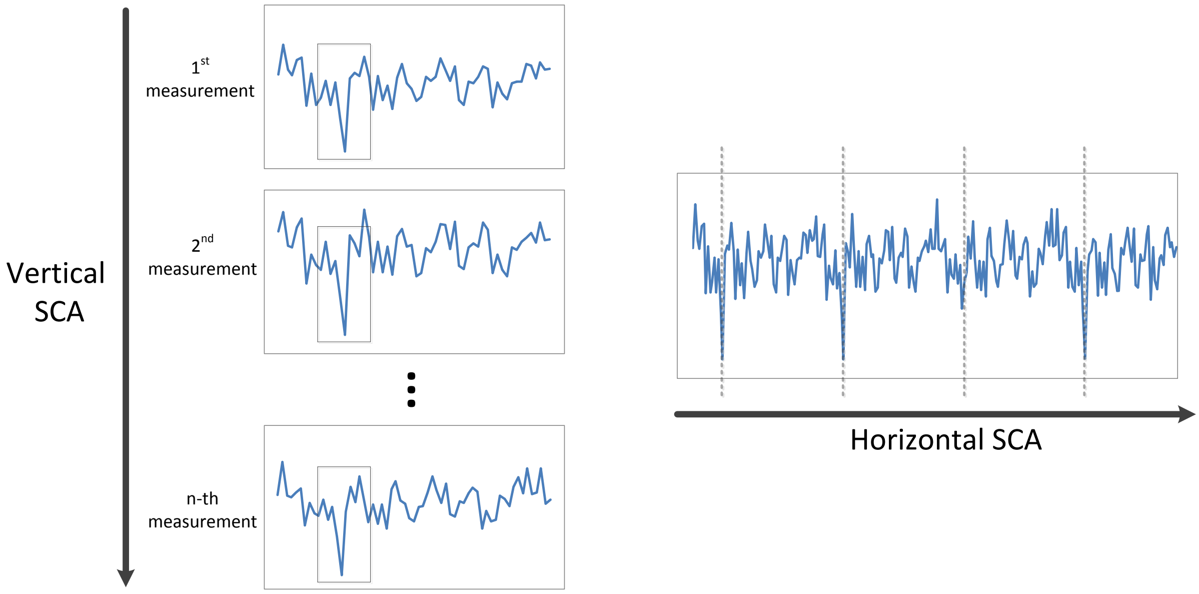


Figure 3.7: Vertical and horizontal SCA attacks.

The kP-operation is executed as a sequence of key dependant iterations of the implemented algorithm. If a measurement is made for a complete execution of the kP-operation, a horizontal SCA attack could be performed by identifying and analysing the single slots in the measured PT. For this reason, a DPA attack can be performed horizontally when analysing an ECC implementation.

There are different statistical methods that can be used for performing a DPA-attack, for example the **difference-of-means test**, the **T-test** [24] and the **Correlation Power Analysis** [25]. The method used for the analyses made in this work was the difference-of-means test, described in the following subsection.

Difference-of-Means Test

For performing a *difference-of-means* test, a PT is first divided into n slots, each corresponding to the processing of a key bit. Each slot is a sequence of power consumption values over a period of time t. Each power consumption value is represented as a point m_j^i, where j is the number of the point within a slot, with $1 \leq j \leq M$; and i is the number of the slot, with

$1 \leq i \leq n$. Each slot consists of the same number of points. For each point m_j^i on every slot i, a point $\bar{m}_j$ with the average value of all points m_j^i is calculated:

$$\bar{m}_j = \frac{\sum\limits_{i=1}^{n} m_j^i}{n}.$$

Consequently, a *mean* curve consisting of all points $\bar{m}_1, \bar{m}_2, \ldots, \bar{m}_M$ is created. The mean curve corresponds then to an average or *mean slot*. Then, the first point of slot 1, m_1^1, is compared to the point $\bar{m}_1$ of the mean slot and, according to this comparison, slot 1 is classified under one of two groups: $S_{key\_bit=0} = \{m_j^i | m_j^i < \bar{m}_j\}$ or $S_{key\_bit=1} = \{m_j^i | m_j^i \geq \bar{m}_j\}$. This means that when slot 1 is classified under $S_{key\_bit=0}$, it is assumed that a key bit with value '0' has been processed for it. When it is classified under $S_{key\_bit=1}$, a key bit with value '1' has been processed for that slot. By doing this comparison with the corresponding point $m_1^2, m_1^3 \ldots m_1^n$ for all further slots $2, 3, \ldots, n$, the first key candidate is obtained. The process is repeated for all remaining points m_{j+1}^i in order to find all other key candidates. The number of key candidates obtained is the same as the number of points on each slot, i.e. M. In Chapter 4 the process of performing a horizontal DPA attack using the difference-of-means test against an IHP ECC implementation is described in detail.

3.1.4 Comparative Power Analysis

ComPA attacks extract the value of a private key by *comparing* two (or more) PTs measured at the same cryptographic device. The main idea is: the processing of the same data during the execution of the same algorithm steps on the same device causes the same power consumption form.

When attacking an implementation of the kP-operation, the same EC point P is selected as input parameter for both cases. One PT is measured when the operation is performed using the *target* private key k, whose value is unknown. The other PT is measured when the private key d is used, whose value is determined by the attacker. The attacker makes a guess for the value of a key bit k_i and sets the guessed value to the corresponding key bit d_i. Then, power consumption measurements are made while the device executes the kP-operation for each case. The two PTs obtained are compared at the corresponding slot i. If the curves of both slots are practically identical, it can be concluded that the same data has been processed at the same slot i, i.e. $k_i = d_i$. This means that the guess for k_i has been done correctly, otherwise the guess was wrong. This process is then repeated for guessing all remaining key bit values until the complete value of k is extracted.

To analyse the correlation of the slots, the difference of the PTs can be calculated. If for both measurements the same data has been processed, both slots are almost equal and their calculated difference is very small [26]. Figure 3.8 illustrates this technique for the case that the same data has been processed during both measurements (left) and the case that different data has been processed (right).

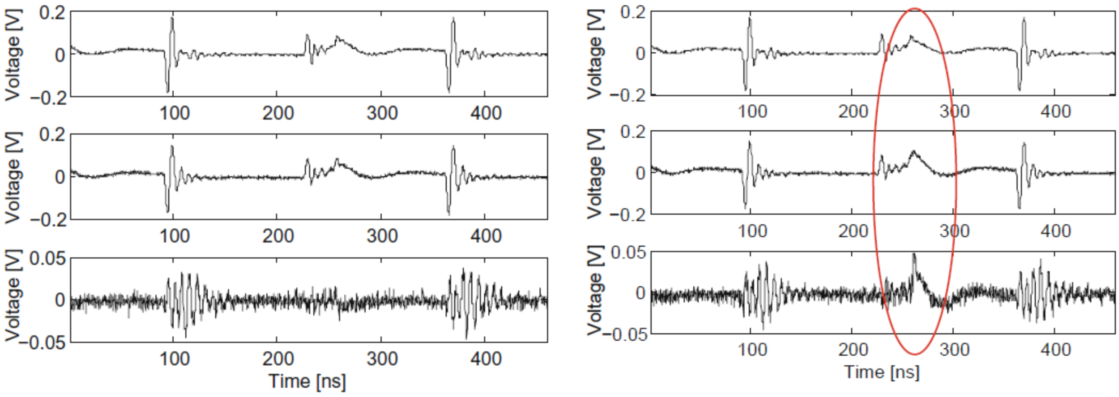


Figure 3.8: Comparative Power Analysis (source: [26]).
The data processing takes place between the 200 and the 300 ns. For the left part of the figure, the difference curve (lowermost) has a value close to zero in this period. For the right part of the figure, the difference curve shows a peak with a much higher value (circled in red).

The attack described in [26] was performed using two separate devices, both identical and consisting of the same cryptographic implementation, but using a different private key each. If the target key is used in a device that does not allow its input parameters to be changed, a second device can be used to perform the cryptographic operations using different key values.

Other kinds of ComPA attacks that can be conducted on implementations of the kP-operation are *collision* attacks. Such attacks are performed on devices that let the attacker determine the value of the EC point P, while k has an unknown value. For each measurement, the value of P is changed.
The work presented in [27] describes an example of collision attacks on implementations based on the double-and-add-always algorithm (see Algorithm 2 on p. 11). It is called the *Doubling Attack* and it requires only two measurements. One PT is measured for the execution of the kP-operation with EC point P as input value. The second measurement is made with point $S = 2P$ as input value. Here, the slot comparison is made between the i^{th} slot measured during the computation of kP and the $(i-1)^{\text{th}}$ slot measured during the

computation of $kS = k(2P)$. For some cases, these two adjacent slots will be very similar: the doubling operations performed on the i^{th} iteration for the computation of kP use the same operands as the one performed during the $(i-1)^{\text{th}}$ iteration for the computation of $k(2P)$. In the double-and-add-always algorithm's main loop, the input value P is doubled $(2P)$ and if the loop iteration has been performed for $k_i = 0$, $2P$ is then used as input for the next iteration. During the next iteration, $2P$ is again doubled $(4P)$. So the collision between the adjacent slots of both traces takes place whenever $k_{i-1} = 0$, where i corresponds to the slot analysed for the computation of kP. For this reason, if this algorithm is executed with the input values k and $(2P)$, the doubling operations performed for every loop iteration for the case $k_i = 0$ will be the same performed on the $(i-1)^{\text{th}}$ iteration of the algorithm's main loop, when executed with the input values k and P. Table 3.2 displays an example for the execution sequence of kP and $k(2P)$ with the double-and-add-always algorithm. Hereby, the four MSBs of k are $(10011...)_2$. It can be seen that each point doubling operation for the processing of key bit k_i in the execution of kP, is the same doubling operation as the one for the processing of key bit k_{i-1} in the execution of $k(2P)$ when $k_{i-1} = 0$.

Table 3.2: Execution sequence of kP and $k(2P)$ with the double-and-add-always algorithm.

k_i	execution of kP	execution of $k(2P)$
1	$2 \cdot 0$ $0 + P$	$2 \cdot 0$ $0 + 2P$
0	$2 \cdot P$ $2P + P$	$2 \cdot 2P$ $4P + 2P$
0	$2 \cdot 2P$ $4P + P$	$2 \cdot 4P$ $8P + 2P$
1	$2 \cdot 4P$ $8P + P$	$2 \cdot 8P$ $16P + 2P$
1	$2 \cdot 9P$ $18P + P$	$2 \cdot 18P$ $36 + 2P$

It can be seen that each point doubling operation processing key bit k_i in the execution of kP, is the same as the doubling operation processing key bit k_{i-1} in the execution of $k(2P)$ when $k_{i-1} = 0$.

The attack presented in [28] identifies similar parts in two PTs at arbitrary slots by using two input messages (EC points) with a more flexible relation than the one used for the Doubling

Attack. This attack is also practicable with a single target device, performing all operations with the private key k.

It is performed under the assumption that a part of the key bits of k have already been extracted, denoted as k_e. k_e bits are the first bits to be processed during the execution of the kP-operation. The rest of the key bits, whose values are unknown are referred to as k_n. An EC point P_Y is selected as the input for the kP operation. Thus, the operation kP_Y is performed and the first PT is measured. During the execution of kP_Y, the first loop iterations are performed processing the key bits k_e. The value of both, the input for each of these loops and the key bit being processed, are known. These are referred to as *reference loops* and its corresponding slots as *reference slots*. Then, a second EC point P_Z is selected as input for performing the kP-operation a second time. Thus, the operation kP_Z is performed and a second PT is measured. P_Z is selected in such a way that when the first loop iteration processing an unknown key bit k_n is performed, the input values for that loop are equal to input values that were processed during the execution of any reference loop. These loops are referred to as *target loops* and their corresponding slots as *target slots*. This means that, if a reference and a target slot are very similar, the same key bit value has been processed for both slots. Thus, the first unknown key bit value k_n can be extracted. This process is then repeated until all unknown key bit values are extracted. Figure 3.9 illustrates this process.

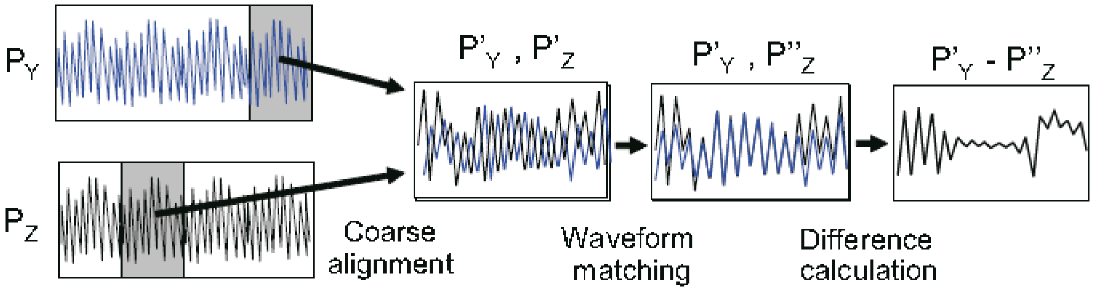


Figure 3.9: Process of a Collision-based attack (source: [28]).
P_Y and P_Z are two different input points. Their relation helps to generate collisions between the PTs measured during the kP-operation execution for each case.

3.1.5 Countermeasures against Differential and Comparative Power Analysis

Implementations protected against SPA can still be attacked using DPA. A well balanced implementation can be successfully attacked with ComPA. A countermeasure based on key- or input randomization makes an implementation more robust against DPA and will also make it more robust against ComPA. These countermeasures were first proposed by Coron in [16].

Key randomization. This countermeasure performs the kP-operation with a randomly computed secret scalar k', but obtains the same outputs as when $Q = kP$ is performed. The method for randomizing (blinding) k consists of adding a multiple of $\#\varepsilon$ to it, where $\#\varepsilon$ is the number of all points on the elliptic curve. So, multiplying any random number r by $\#\varepsilon$ and adding it to k results in k'. And $k'P = kP$ since $\#\varepsilon P = \boldsymbol{O}$:

$$k' \cdot P = (k + r \cdot \#\varepsilon) \cdot P = kP + r \cdot (\#\varepsilon P) = kP + r \cdot \boldsymbol{O} = kP.$$

Base point blinding. The randomized value in this case is the point P. The point R is randomly chosen and added to P, the scalar multiplication is then performed as $Q = k(P + R)$. After Q is computed, the known value $S = kR$ is subtracted from it, giving $Q = kP$. The parameters R and S are initially stored and updated at each performance of the kP-operation[3]. The attacker doesn't know that the point $P' = P+R$ is being multiplied by k.

Further countermeasures are: randomizing the projective coordinates [16]. Here, the projective representation of a point P is randomized before each new execution of the kP-operation. Another countermeasure is randomly splitting the secret key into $k = k_1 + k_2$, described as well in [29]. Randomizing the register address to disconnect the relation between key bits k_i and register addresses was proposed in [30]. The work described in [31] proposes and presents an analysis of randomizing the steps of the implemented kP-algorithm as a strong countermeasure against DPA.

Each countermeasure has been proposed to make an implementation resistant against a specific attack. Nevertheless one countermeasure may simplify another attack. The awareness of different types of attacks can lead to a more secure cryptographic implementation when the countermeasures against these attacks are taken into consideration.

[3] This means that the additional EC point multiplication $S' = kR'$ using the random point R' has to be pre-computed for each new kP-operation. To save time and energy consumption, $S = kR$ can be pre-computed and the random point can then be calculated as follows: $R' = 2R, S' = 2S, R'' = 2R', S'' = 2S'$; and so on.

3.2 Fault Sensitivity Analysis

Fault Injection (FI) attacks are based on analysing the changes on the behaviour of cryptographic systems when faults are induced in them. Such attacks are classified as active attacks since the functionality of the devices is manipulated [32].

Generally, the attacked implementation has to be able to output faulty results in order to be analysed with this method. The Fault Sensitivity Analysis (FSA) attacks do not need the faulty output and they can be applied to cryptographic devices with implemented countermeasures against FI, which stop a device from delivering faulty results as outputs. In FSA an operation's sensitivity to FI is analysed. The faulty outputs of the operations are not analysed, but rather only if these operations are performed correctly or not. Therefore, this section describes shortly the process of fault injection and the technique used to perform this attack.

3.2.1 Fault Injection

The main purposes of inducing faults are either to hinder an instruction execution from taking place or to manipulate the data being processed. The most common technique for performing this is through a *clock glitch*, illustrated in Figure 3.10. The frequency of the clock coordinating the activities in the cryptographic device's circuit is altered at a precise moment. This causes a setup time violation during the glitch cycle and an error for a specific calculation can be introduced.

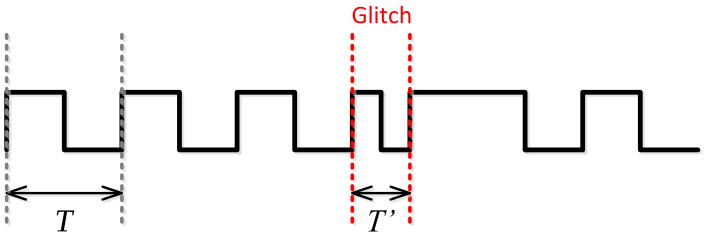


Figure 3.10: Clock glitch.
Two types of clock, one with period T and a faster one with period $T' < T$, generate an *illegal clock*. $f = 1/T$ is the maximal allowed clock frequency in the circuit for producing correct outputs during all cycles. In the cycles at which an error should be induced, the frequency is increased to $f' = 1/T'$ and a glitch is caused.

The frequency needed for FI varies for each clock cycle since it is dependent on the operations performed and data being processed during each clock cycle. Variations of this *threshold condition* are seen, for example, during the execution of a cryptographic algorithm. A small acceleration of the clock can induce faults during the cycles at which arithmetic operations are performed. On the other hand, a bigger acceleration of the clock's frequency is needed to induce faults during the cycles at which data is only being stored. The higher a frequency needs to be for FI, the lower is the Fault Sensitivity (FS) of the circuit at this clock cycle.

3.2.2 FSA Attack Technique

The basic assumption in PA attacks is that whenever an operation is performed using the same inputs, the same energy is consumed by the cryptographic device. For FSA attacks the assumption is that a system has the same FS whenever one operation is performed using the same input data. The FSA attack uses a circuit's FS as leakage source.
An attacker analyses the threshold conditions during the execution of the point doubling operations in a chip executing the Montgomery kP-algorithm (see Algorithm 3 on p. 12). The point doubling operations are performed once during each iteration of the inner loop. If the iteration is running for $k_i = 0$, then the EC point stored in $Q[0]$ is doubled. If it is running for $k_i = 1$, then $Q[1]$ is doubled. Table 3.3 shows a calculation sequence example for the input point P and the private key $k = (1101)_2$. The underlined values are the results from the doubling during the iteration.

Table 3.3: Sequence of point doublings and point additions for an EC point multiplication.

i	3	2	1	0
k_i	1	1	0	1
$Q[0]$	P	$3P$	$\underline{6P}$	$13P$
$Q[1]$	$\underline{2P}$	$\underline{4P}$	$7P$	$\underline{14P}$

It is important to remind that in the Montgomery kP-algorithm the Most Significant Bit (MSB) of k always has the value '1'. For this reason, a point doubling operation for $Q[1]$ is always performed at the beginning of the kP-execution during the initialization phase. This calculation can be referred to as the *initial doubling*.

A key extraction can be successfully performed when it is possible to distinguish, which input data has been used for performing the point doubling in each iteration. If a device performs two point doublings using the same input data for each operation, the FS of the device while performing these operations is very similar for both cases.

The input points for the initial doubling of the Montgomery kP-algorithm can be selected by the attacker. If the point P is selected as input, it is known that the initial doubling will be $Q[1] = 2P$. If the point $2P$ is selected as input, the initial doubling operation will be $Q[1] = 2 \cdot 2P = 4P$. This way, the attacker can perform an FSA on the device while it performs doubling operations with selected data as input. Any input value can be selected for the initial doubling. It can be observed how the circuit reacts to FI while performing this operation with different inputs. Figure 3.11 shows an example of the results obtained from such an analysis. The diagram displays the results of an FSA for the execution of a point doubling operation, conducted with an initial frequency of 23 MHz. The frequencies with which a fault injection occurs for each clock cycle are shown as points. The points at 28 MHz mean that no faults were injected during these clock cycles.

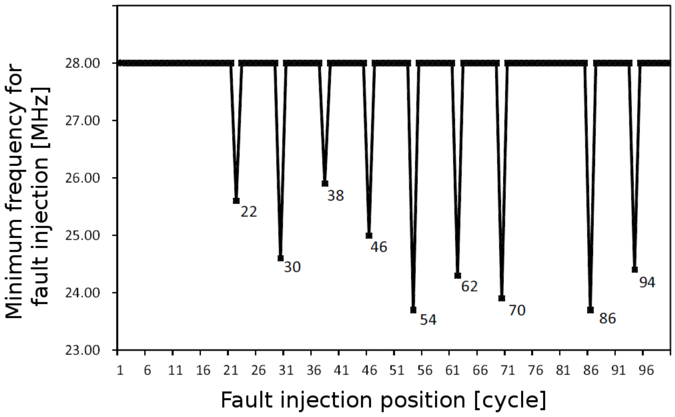
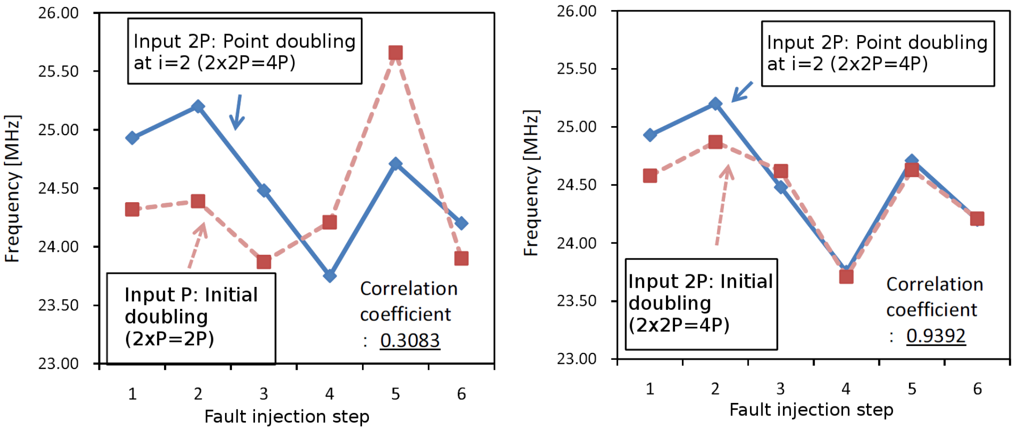


Figure 3.11: Fault Sensitivity Analysis (source: [19]).
The results of an FSA for the execution of a point doubling operation are displayed. The operation was initially executed with a frequency of 23 MHz. The frequencies with which an FI occurs for each clock cycle are shown as points. The points at 28 MHz mean that no faults were injected during these clock cycles.

The results of the analysis made during the initial doubling can then be used as a template, i.e. they can be compared to FSAs performed for the further iterations of the algorithm. For each iteration of the kP-operation, the attacker performs an FSA and compares the results to analyses made with the different inputs for the initial doubling. According to this comparison, the attacker is able to define what input has been used for the doubling operation and thus, which key bit has been processed during the iteration.

An example of this process can be described based on the execution sequence shown in Table 3.3. The initial doubling $2P$ is performed for the processing of key bit k_3 and the first loop iteration is performed processing k_2. The attacker performs an FSA for the doubling of this iteration and compares it to the FSA performed for the initial doubling. In this case, the results of the analysis do not correlate: for the initial doubling, point P is used as input and for the doubling in the first loop iteration $2P$ is used as input. This means that two different point doublings are performed: $2 \times P$ and $2 \times 2P = 4P$. Therefore, a comparison is made with the FSA performed for the initial doubling with input $2P$. This time, both analyses show a high correlation and it is concluded that $k_2 = 1$. This process is then repeated and further key bits are extracted. Figure 3.12 illustrates how such a comparison can be done. Hereby, (a) shows a comparison of FSA results that correspond to different point doublings and (b) shows a comparison made when the analysis results correspond to the doublings performed with the same input data.



(a) Low correlation of the FSA results

(b) High correlation of the FSA results

Figure 3.12: Fault Sensitivity comparison (source: [19]).

3.2.3 Countermeasures against Fault Sensitivity Analysis

In order to perform this analysis, the attacker needs to provide EC points as inputs for the cryptographic device being analysed. Consequently, any ECC implementations that do not allow this are protected against this attack. All DPA and ComPA countermeasures, such as randomisation of the EC point, key or coordinates (see section 3.1.5), hinder this attack.

Due to the big number of countermeasures that can stop the FSA attack from taking place and the extensive measurements that are needed for it, its application for a private key extraction is not very practical in comparison for example to ComPA attacks. Nevertheless the concept of analysing the execution of initial operations and using these analyses as templates can become useful for further attacks. To prevent this, the initial doubling can be implemented in a different way as it is implemented for the main loop iterations. This way, the power consumption or FS templates made with the initial doubling are not useful for comparison with the rest of the doublings performed during the execution of the Montgomery kP-algorithm. This idea is described in section 5.2 (see p. 92).

The next Chapter describes the implementation details of the IHP ECC design. This ECC design was resistant against SPA but not against other SCA attacks described in this chapter. The implementation details are described as part of a vulnerability assessment of the design.

4 Vulnerability Assessment of an IHP ECC Design

When a cryptographic implementation can be successfully attacked, it is necessary to understand the attack's and the implementation's details in order to discover security gaps in the design that have lead to the successful performance of the attack. This way, countermeasures against it can be applied or developed in order to increase resistance of the implementation against this attack and thus make the implementation more secure.

This chapter presents a vulnerability assessment of an IHP ECC design. The implementation's details are described in section 4.1. A description of the DPA performed on this design using the difference-of-means test is given in section 4.2. Section 4.3 provides information about the system's architecture, the system's control signals and a detailed description of the implemented Montgomery kP-algorithm.
Observations that lead to the implementation's vulnerabilities are listed before concluding this chapter with suggestions on how to improve this design regarding its protection against SCA attacks.

4.1 IHP ECC Design

The IHP hardware accelerator for the ECC is an implementation of the Montgomery algorithm for elliptic curves point multiplication kP (see Algorithm 4 in p. 13). The following points describe further characteristics of the implementation:

- The algorithm is implemented for NIST elliptic curves B-233 [8].
- The implementation is optimized for IHP 130 nm technology.
- Points of the EC are represented using Lopez-Dahab projective coordinates [18].
- The $GF(2^{233})$ element's multiplication has been implemented using the *4-segment iterative Karatsuba* multiplication method [33] and needs 9 clock cycles for calculating one product.

- For each partial multiplication, the 233-bit long operands are fragmented into four 59-bit long segments.
- The analysed version of the ECC design is a further developed version of the one described in [34]. In comparison to the original version described in [34], the analysed design is resistant against SPA.
- Countermeasures against DPA had not been implemented in this design at the time this assessment was performed. A DPA attack using the difference-of-means test was successfully realized as part of the TAMPRES-Project [35].

4.2 Difference-of-Means Test

This section describes how a horizontal DPA attack using the difference-of-means test was performed against the IHP ECC design. The analysis has been carried out based on simulated values of the power consumption of the design while executing the EC point multiplications kP (from this chapter on referred to as kP-operation).
The difference-of-means test was performed using simulated PTs for two different cases:

- case 1: for the kP-design processing the EC point $P1 = (x_1, y_1)$ as the input data using the scalar $k1$
- case 2: for the kP-design processing the EC point $P1 = (x_1, y_1)$ as the input data using the scalar $k2$

The following values were assigned to the EC point coordinates and scalars, here represented in hexadecimal notation:

x_1 = 181 856adc1e 7df13784 91fa736f 2d02e8ac f1b9425e b2b061ff 0e9e8246

y_1 = 89 fed47b79 6480499c baa86d8e b39457c4 9d5bf345 a0757e46 e2582de6

$k1$ = 93 919255fd 4359f4c2 b67dea45 6ef70a54 5a9c44d4 6f7f409f 96cb52cc

$k2$ = cd ea65f6dd 7a75b8b5 133a70d1 f27a4d95 06ecfb6a 50ea526e b3d426ed

Simulation results of the power consumption of the investigated IHP ECC design were obtained using the Synopsis PrimeTime suite [36] and saved in an ".out"-file. This file consists of the power consumption not only for the complete design, but also for each of its entities, i.e. for each individual block of the ECC design. The difference-of-means test was performed for the following blocks separately:

1. complete ECC design (ecc)
2. block Multiplier (mult)
3. block Arithmetic Logic Unit (ALU) (alu)
4. register X1 (x1)
5. register Z1 (z1)
6. register X2 (x2)
7. register Z2 (z2)

The names written in brackets correspond to the names used to describe the results of the test regarding each block. Additionally, the difference-of-means test was performed for the sum of the power consumptions of the registers X1, Z1, X2, Z2 and the ALU block (all_register+alu). The analysis of individual blocks of the ECC design can be useful for uncovering which of these blocks are SCA leakage sources and how.

Each simulated PT of the investigated IHP ECC design can be separated in parts corresponding to the kP-algorithm:

- **The initialization part:** This part corresponds to the initialization phase of the Montgomery kP-algorithm. During this phase, the conversion of affine EC point coordinates (x, y) to projective coordinates (X,Y,Z) takes place and the most significant bit of the scalar k is processed. This part takes 8 clock cycles.

- **The part of the processing of all remaining bits of the scalar (i.e. key) $\boldsymbol{k}$, excluding its most significant bit:** Both scalars that are used – $k1$ and $k2$ – are 232 bit long. This means that this part consists of 232-1=231 slots[1]. The duration of each slot is always 57 clock cycles. One single power value is present during each clock cycle[2], consequently each slot consists of 57 points. So, the kP-operation needs $(232 - 1) \times 57 = 13167$ clock cycles to be calculated.

- **The last part of the $\boldsymbol{kP}$ trace:** This part corresponds to the conversion of the multiplication result $kP = (X, Y, Z)$ back to affine coordinates and takes 431 clock cycles.

For the difference-of-means test, only the part of the kP-operation that corresponds to the processing of the scalar was chosen, i.e. the part with 231 slots. The test was performed as follows:

[1] Each slot is a part of a PT which corresponds to the processing of only one bit of the scalar k, i.e. one execution of the algorithm's main loop.

[2] Simulation setup.

1 - The investigated part of the PT was partitioned into 231 slots, each one consisting of 57 clock cycles. Each slot was represented as a curve consisting of 57 points. Figure 4.1 shows the first 8 slots of the simulated PT for case 1. Figure 4.2 shows the slots from Figure 4.1 clockwise in the same coordinates. This way, the points of each slot can be compared with corresponding points of the other slots.

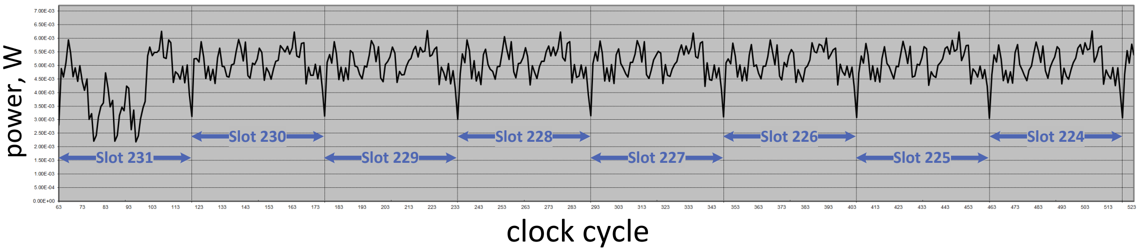


Figure 4.1: First 8 slots of the simulated PT for case 1.
The IHP kP-design processed the EC point $P1 = (x_1, y_1)$ as the input data using the scalar $k1$.

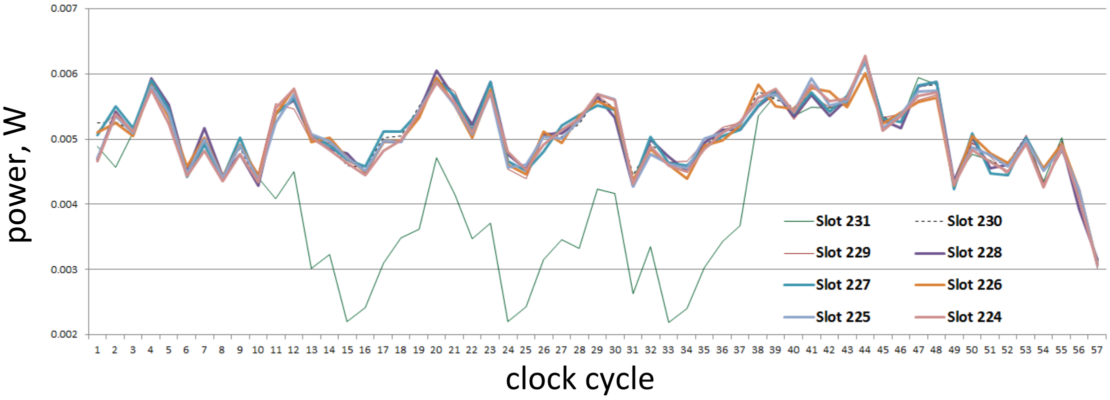


Figure 4.2: All slots from Figure 4.1 shown clockwise in the same coordinates.

2 - The mean curve of all 57-point-long curves was then calculated.

3 - The power value of the 1st point of the mean curve was compared with the power value of the 1st point of each slot. The first comparison was made with slot 231, which corresponds to the processing of the key bit $k1_{230}$. The last comparison was made with slot 1, which corresponds to the processing of the key bit $k1_0$. If the power value of the mean curve was higher than the value of the slot compared, it was assumed that the slot corresponded to the processing of a key bit with value ‘1’. Otherwise it was assumed that the slot corresponded to the processing of a key bit with value ‘0’. Thus,

the first key candidate was obtained based on comparisons of the 1st points of all slots with the 1st point of the mean curve.

4 - Step 3 was repeated for all other 56 points of the mean curve and the remaining 56 points of each slot. So, the next 56 key candidates were obtained.

Usually, the *variance* is calculated to obtain information about SCA leakage sources. In this work it was calculated for each point j, with $1 \leq j \leq 57$, of the mean curve as follows:

$$variance_j = \sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n}(x_j^i - \bar{x}_j)^2.$$

Here, $n = 231$ is the number of slots and $\bar{x}_j$ is the value of the j^{th} point of the mean curve, i.e. the average value of all point values x_j^i of all slots i.

Using this formula, the variances for all 57 points of the mean curve were calculated. Figure 4.3 shows all calculated variances as a curve consisting of 57 points.

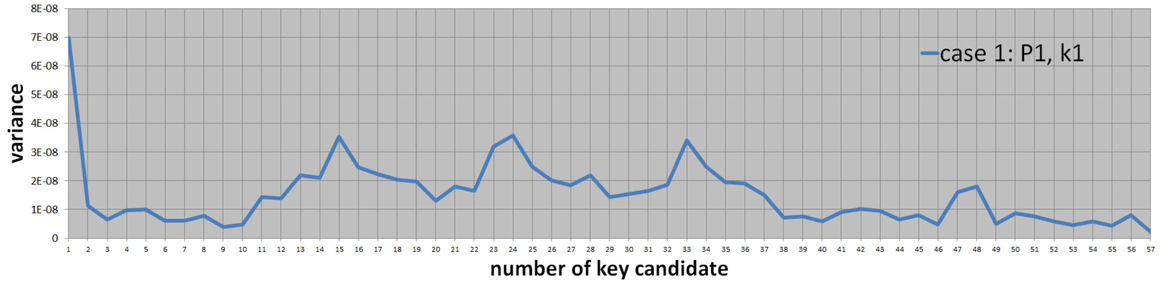


Figure 4.3: Variances as a 57-point-long curve for case 1.

The peaks on the variance curve are usually defined as potential SCA leakage sources. Applying this approach, at least 4 SCA leakage sources can be detected: they are the activities performed at clock cycles 1, 15, 24 and 33 of the slots.

The variance curve for case 2, i.e. for the same ECC design processing the same EC point but using scalar $k2$ as key is shown in Figure 4.4.

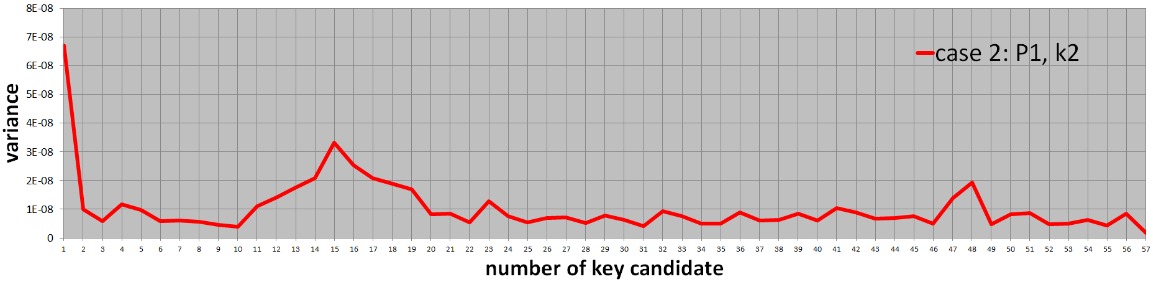


Figure 4.4: Variances as a 57-point-long curve for case 2.

Only three peaks are shown in the variance curve in Figure 4.4: at clock cycles 1, 15 and 48. Both calculated variance curves differ significantly from each other.

Since the variance curve cannot be used to define clearly all SCA leakage sources of the investigated design, each one of the 57 key candidates was compared with the key bit value that was processed. For each key candidate, three different comparisons were made:

Comparison model v1: This is a direct comparison of the key candidate's bit value with the key bit value that was actually processed for that slot. The suggestion is that the bit value of the key candidate extracted using slot i corresponds to the bit value k_i.

Comparison model v2: A comparison of the key candidate's bit value extracted using slot i with the key bit value k_{i-1}, i.e. the key bit value that was processed for the corresponding *previous* slot.

Comparison model v3: A comparison of the key candidate's bit value extracted using slot i with the key bit value k_{i+1}, i.e. the key bit value that was processed for the corresponding *following* slot.

These three comparisons were taken into consideration due to the fact that, even though each slot corresponds to the processing of one single bit, the values in the curve of the slot could be influenced by the key bit processed for the previous or the following slot. This could take place for example if the last operations of the previous loop iteration are executed during the first clock cycles of the actual iteration. In a similar way, operations belonging to the following iteration could already start being executed during the last clock cycles of the actual iteration. The attack model using the difference-of-means test was extended with these three comparison suggestions.

With this approach, three different key candidates were obtained for each point of the mean curve, so a total of 3×57 key candidates were obtained. The relative correctness of each key candidate was calculated as $number\_of\_correct\_extracted\_bits/231 * 100\%$[3]. This relative correctness of the extraction of the key was presented for the three variants of each of the 57 key candidates as 3 curves consisting of 57 points each (see the white dotted curve for comparison model v1, the black dotted curve for v2 and the yellow dotted curve for v3 in Figure 4.5).

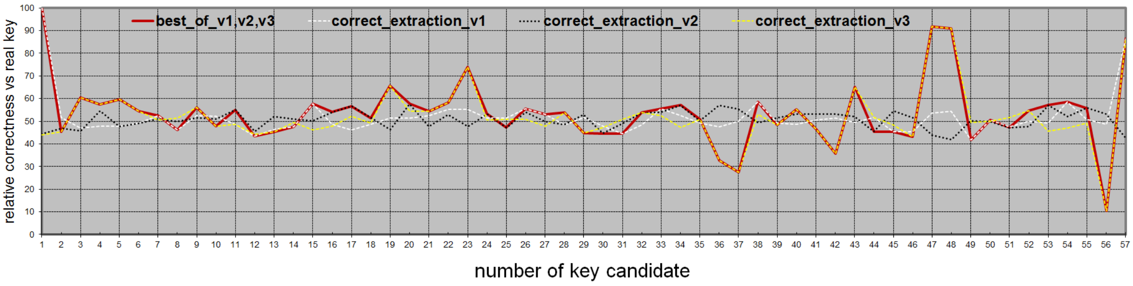


Figure 4.5: Relative correctness of the extraction of the key for each of the 57 key candidates for each of the three comparison models.

From the security point of view the ideal case is if the correctness of the key extraction is 50% for all key candidates. This would mean that only 50% of all key bits have been correctly extracted. From the attacker point of view, the best result is the one with the relative correctness that is more distant from 50%, for example 100% or 0%. If the relative correctness of the key extraction is 0%, it means that all bits of the key candidate have been extracted wrong. The inversion of all key bits of this key candidate would give a new candidate with 100% correct extracted key bits. In this context, the red curve in Figure 4.5 displays the best attack results out of the three comparisons.

The correctness of the key extraction for the key candidates 1, 47, 48 and 57 is extremely high (see red curve in Figure 4.5). For example, the correctness of the key candidate 1 is 100%. Contrary, the correctness of the key candidate 56 is extremely low, only about 15 %. This means that the basic assumption for the search of the key candidate is wrong. But it means that, for example, for the 56th point of each slot the "inverted" assumption is correct: if the power value of the 56th point of the mean curve is higher than the value of the current slot, it corresponds to the key bit value '0', otherwise to '1'. The new key candidate, that was obtained using this new assumption, is equal to the inverted "old" key candidate. The correctness of this new key candidate number 56 is $100\% - 15\% = 85\%$. The relative correctness of the extraction of the key for each of the 57 key candidates using the inverted assumption in comparison to the "old" assumption is shown in Figure 4.6.

[3] $n = 231$ is the number of slots for this difference-of-means test.

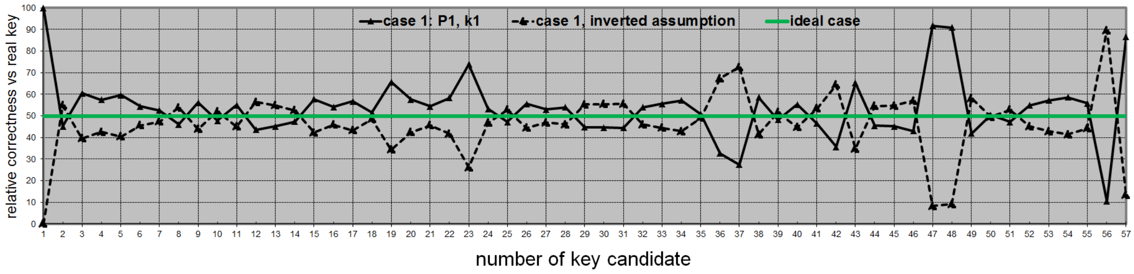


Figure 4.6: Relative correctness of the extraction of the key for each of the 57 key candidates using the “direct” and “inverted” assumptions.

The difference-of-means test with three comparison models was done for case 2 as well, in which the kP-operation was executed using the same EC point $P1$ but with a different key $k2$. The black curve in Figure 4.7 shows the relative correctness of the extraction of $k2$ (best results out of the three comparison models). The red curve represents the best results obtained from the key extraction of $k1$; this is the same red curve shown in Figure 4.5. The green curve corresponds to the ideal case.

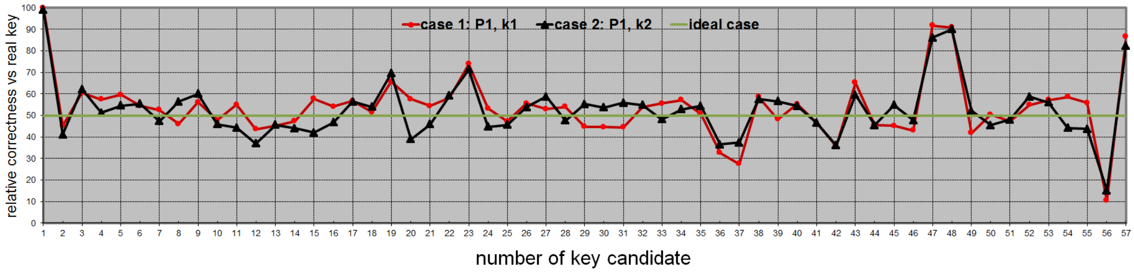


Figure 4.7: Relative correctness of the extraction of the key for each of the 57 key candidates for both analysed cases.
The red curve corresponds to the case 1 and the black curve to the case 2; the green curve shows the ideal case.

The correlation between the red and the black curves in Figure 4.7 is much higher than the correlation between the variance curves (see Figure 4.3 and Figure 4.4) for both cases: in both investigated cases the correctness of the key extraction for the key candidates 1, 47, 48, 56 and 57 is extremely high. The variance curves for both cases gave only clock cycle 1 as an SCA leakage source.

Using the relative correctness of the key extraction for the two investigated cases, the following can be summarized: 5 out of 57 calculated key candidates - numbers 1, 47, 48, 56 and 57 - are extracted with a correctness from 85% up to 100%. The next 3 key candidates – number 19, 23 and 37 are extracted with a correctness of about 70%. With these results, the following points can be summarized for the IHP implementation of the kP-operation:

- the implementation can be successfully attacked using the difference-of-means test by a horizontal DPA attack,
- the implementation has many SCA leakage sources, the most relevant are found at clock cycles: 1, 47, 48, 56 and 57 of the main loop iteration of the implemented algorithm.

The analysis of the results of the difference-of-means test using relative correctness curves provides many more correct defined SCA leakage sources in comparison to the calculation of the variance. Cryptographic designers can always calculate the relative correctness since they know the actual value of the processed key. Attackers on the other hand, can only do this for implementations that allow a value of k to be given as input; or for implementations with key randomization methods that are manipulable. This can be useful for estimating the design's robustness against SCA before its actual hardware implementation takes place, which can save costs in production.

The difference-of-means test performed with the PT of each individual block of the ECC design shows many more possible sources of SCA leakage. Figure 4.8 shows the relative correctness of the extraction of the key for each of the 57 key candidates for blocks of the ECC design, it consists of the following figures:

a) The relative correctness curve for the complete ECC design (ecc) is displayed in black colour. The red curve corresponds to the block Multiplier (mult).

b) The relative correctness curve for register X1 (x1) is displayed in blue. The orange dotted curve displays the relative correctness of register Z1 (z1).

c) The relative correctness curve for register X2 (x2) is displayed in blue. The white dotted curve displays the relative correctness of register Z2 (z2).

d) The relative correctness curve for the ALU block (alu) is displayed in black.

e) This curve corresponds to the relative correctness calculated when all four registers X1, Z1, X2, Z2 and the ALU are considered as one block (all_register+alu), i.e. the analysed PT was calculated as the sum of the PTs of these five blocks.

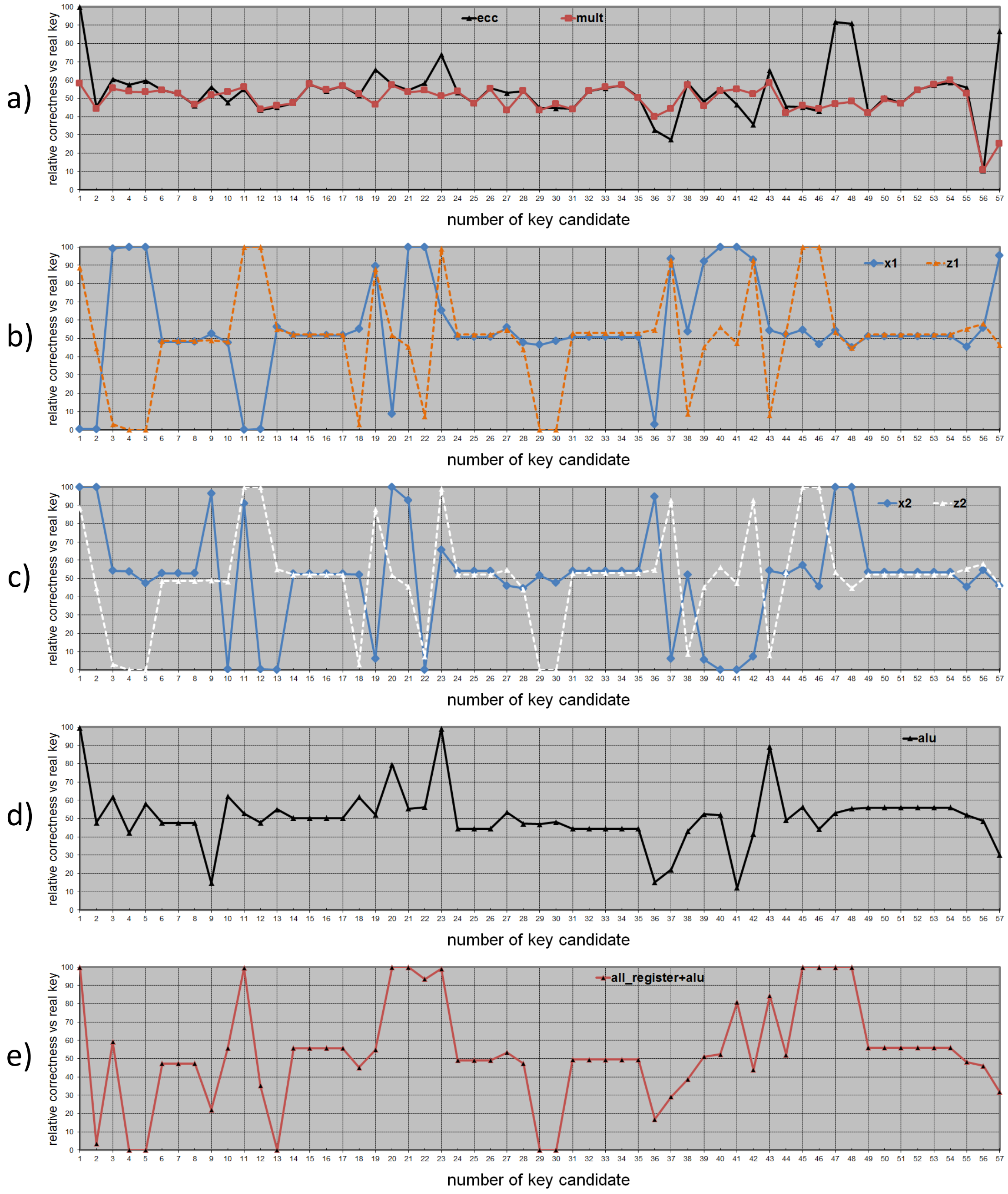


Figure 4.8: Relative correctness of the extraction of the key for each of the 57 key candidates using PTs of the ECC design and its individual blocks separately.

The registers X1, X2, Z1, Z2 and the block ALU are the most insecure (leaking) blocks (see Figure 4.8 b), c) and d)). By considering the activities of all registers together and the ALU as one block (Figure 4.8 e)) less leakage sources can be observed. The activity of the block Multiplier (see Figure 4.8 a)) can be assessed as a significant SCA leakage source at points 56 and 57 only. The power consumption of the Multiplier is very high compared to other blocks. Its constant activity *covers* leakage sources caused by other blocks. For this reason, if the Multiplier is inactive[4], the activity of other blocks is easier observable.

4.3 Technical Description

In this section, the implementation of the IHP ECC design is described based on an analysis made on its source code. The analysis focuses on the main loop of the Montgomery kP-algorithm, which corresponds to the slots used for the difference-of-means test. Special focus is put on the parts of the loops executed during the clock cycles for which a key extraction was made with a high relative correctness in the difference-of-means test. This way, the causes of the successful key extraction could be defined.

4.3.1 System Architecture

The analysed IHP ECC design was implemented in the hardware description language Very High Speed Integrated Circuit Hardware Description Language (VHDL) and consists of 17 files describing registers and mathematical operators. Each Functional Unit (FU) in the design consists of one VHDL file. The value of the key k and the coordinates of the EC point $P = (x, y)$ as input parameters were included in an additional test bench file. The inputs k, x, and y can be up to 233-bit long binary numbers.
For the analysis, P is the point $P1$ which was used as input for performing the difference-of-means test (see p. 38 on section 4.2). The key k is only 10 bit long: $k = 2cc$ (in hexadecimal). This length was selected to reduce the simulation time and simplify the analysis. The simulated traces for the power consumption of the entire design and of its individual FUs were generated with the software PrimeTime [36].

[4] The functionality of the Multiplier will be explained in section 4.3.

The system architecture of the ECC design is represented by the block diagram shown in Figure 4.9.

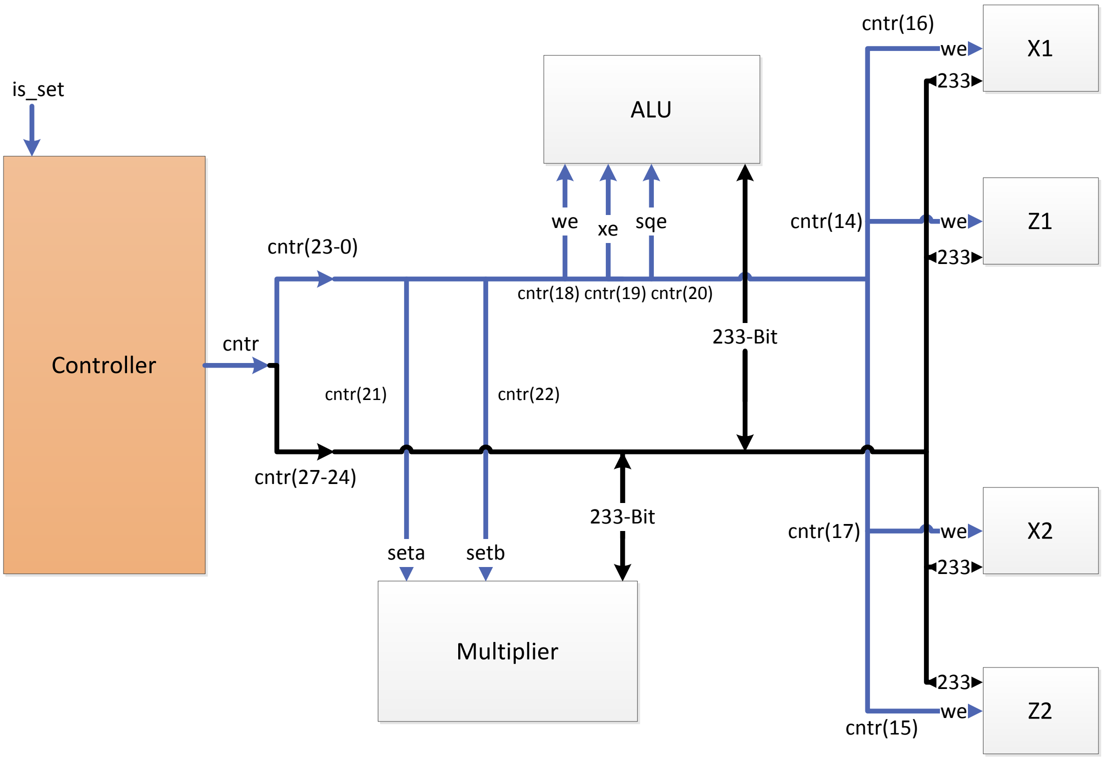


Figure 4.9: Structure of the IHP ECC design.
The entities shown in this diagram execute the kP-operation and are connected with each other through a bus channel. The FUs X1, Z1, X2 and Z2 are 233-bit long registers. The FUs Multiplier and ALU perform mathematical operations in $GF(2^{233})$. The Controller controls the complete calculation process.

Each FU shown in Figure 4.9 is described below:

- **Controller:** This FU is described as a 32-bit register and it manages the work of all other FUs. The bits 0-27 of this register manage the access of all other FUs to the Bus. This way, the execution process of the kP-operation is controlled. Figure 4.9 shows the single bits of the `cntr` signal next to their corresponding inputs on the other FUs. The input signal `is_set` determines if a key bit with value ‘1’ or with value ‘0’ should be processed. Depending on this and on its current internal status, the value of the output signal `cntr` is set. A detailed description of `cntr` is presented in section 4.3.2.

- **Bus channel:** This entity is responsible for the data exchange between all other FUs in the design. The value on its input signal `cntr(27-24)`, bits 27 to 24 of `cntr`, determines the FU who's output data is to be written to the Bus. This data is transmitted to the inputs of all other FUs immediately. The Controller decides which of these FUs should save this data in their internal register. All data inputs and outputs are connected this way through the Bus.

  Additionally during the execution of the kP-operation the Bus reads once the output data from the registers x[5] and b[6] of the ECC design. These two registers are not shown in Figure 4.9.

- **external registers X1, Z1, X2, Z2:** These entities are 233-bit long storage registers. If their input signal `we` has the value '1', the data from the Bus will be saved in the register. These registers are used to save intermediate values during the execution of the Montgomery kP-algorithm.

- **ALU:** The ALU has a 233-bit long data input and a 233-bit long data output. The results of its operations are saved in an internal 233-bit long register. All values saved in this register are automatically sent to the ALU's output. The ALU is responsible for two arithmetic operations: addition and squaring of the elements of $GF(2^{233})$.

  When the ALU's input signal `we` has the value '1', the data on its input (that is, the data from the Bus) is saved in the internal register. When the signal `xe` has the value '1' the data on its input is added (*XOR*ed) to the data saved in the internal register. This means that the addition of two elements of $GF(2^{233})$, that is the bitwise *XOR* of two big binary numbers, needs two clock cycles to be executed.

  When the ALU's input signal `sqe` has the value '1', the data on its input is squared and saved in its internal register.

- **Multiplier:** When the input signal `seta` has the value '1', the Multiplier saves the data from the Bus as the first operand in one internal register. The second multiplicand is saved from the Bus in another internal register when the signal `setb` has the value '1'. Once both operands have been saved, the multiplication starts. It is finished after 9 clock cycles. This means that the Multiplier needs in total 11 clock cycles in order to complete a multiplication: 2 cycles for saving both operands and 9 cycles for calculation of one product.

[5] Register x saves the x-coordinate of the EC point $P = (x, y)$ during the execution of the kP-algorithm.

[6] Register b contains the standard value b = 066 647ede6c 332c7f8c 0923bb58 213b333b 20e9ce42 81fe115f 7d8f90ad [8]. It is one standard parameter of the EC B-233.

4.3.2 Control Signal `cntr` Description

The block diagram in Figure 4.9 shows how the FUs in the analysed design are controlled using the Controller's output signal `cntr`. This is an input signal for all other FUs, controlling the Bus access to their inputs and outputs. This subsection describes the configuration of this signal and the function of its single bits.

Depending on its internal state, the Controller sets the value of this signal and sends it as separated bits to all other functional units of the ECC design. Figure 4.10 illustrates the configuration of the `cntr` signal, showing its single bits and the corresponding names of the FU's inputs to which they are sent.

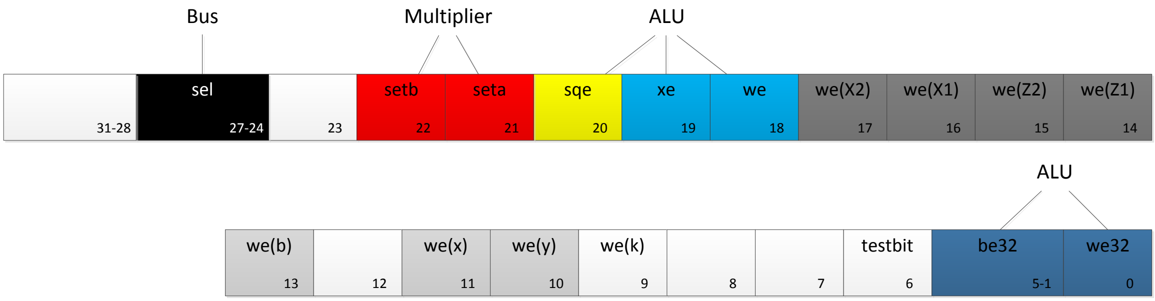


Figure 4.10: Configuration of the signal `cntr`.

The bits 31-28, 23, 12, 8, and 7, represented in white boxes in Figure 4.10, are unused. The functionality of all used bits and bit groups of the signal `cntr` is described below:

- **Bits 27 to 24:** This group of bits — a 4-bit long word — is the input data of the Bus. The value of this bit word indicates which FU's data output should be written to the Bus. The values represented as decimal numbers in Table 4.1 determine the Bus access to the output data of the corresponding FUs. The names given in the table for the signals are the names used in the design's source code (file `ecc_233.vhd`). These are the names used for the port mapped signals between the functional units in the design.

Table 4.1: Description of the input values for the Bus.

`sel`, **Used Value**	**Signal Name**	**Description**
0	`o_ext_reg_r_out`	output of external regiser is written to the Bus
1	`o_b_r_out`	output of register b is written to the Bus
2	`o_z1_r_out`	output of register Z1 is written to the Bus
3	`o_z2_r_out`	output of register Z2 is written to the Bus
4	`o_x1_r_out`	output of register X1 is written to the Bus
5	`o_x2_r_out`	output of register X2 is written to the Bus
6	`o_multiply_result`	output of Multiplier is written to the Bus
7	`o_alu_r_out`	output of ALU is written to the Bus
8	`o_x_r_out`	output of register x is written to the Bus
9	`o_y_r_out`	output of register y is written to the Bus

The Bus receives four bits from `cntr`, bit 27 to bit 24, as the input signal `sel`. The binary number represented by this 4-bit word determines the access control of the Bus. This means that depending on the value of `sel`, the output from one FU of the design is written to the Bus as input data for other FUs.

- **Bits 22 and 21:** These bits are sent to the inputs `seta` and `setb` of the FU Multiplier respectively. If any of these bits has the value '1', the Multiplier saves the value from the Bus in one of its internal registers and uses it as one of the multiplicands for the next multiplication. The multiplication starts as soon as both values have been set.

- **Bits 20, 19, 18:** These bits are sent to the inputs `sqe` (square enable), `xe` (xor enable), and `we` (word enable) of the FU ALU respectively. The value on these inputs determines the operation to be executed by the ALU with the data from the Bus on the current clock cycle: This way input data of the ALU can either be squared (`sqe`) and saved in the ALU's internal register, only saved (`we`) in the internal register, or added (`xe`) to the value already saved in the internal register. Everything saved in the ALU's internal register is automatically placed on the ALU's output.

- **Bits 17, 16, 15, 14:** These bits control the functionality of the registers X2, X1, Z2, and Z1 respectively. For each bit with value '1', the corresponding register saves its data input at the current clock cycle. This way, values can be saved within the process.

- **Bits 13, 11, 10, 9:** These bits control the functionality of the registers b, x, y, and k respectively. For each bit with value '1', the corresponding register saves the actual value located on its data input at this clock cycle.
  These bits are only used outside the Montgomery kP-algorithm's main loop executions for specific purposes. The value of bit 13 is never set to '1'. Bits 11 and 10 are used only once each at the end of the kP-operation to overwrite the registers x and y with its final output values. Bit 9 is used regularly in the finalization phase after all loops in the algorithm have been processed.

- **Bit 6:** This bit controls the functionality of the test bit register. When this bit has the value '1', the test bit register saves the actual value located on its input data at this clock cycle. At the same time this value is placed on its output, which is sent to the Controller as the `is_set` signal.

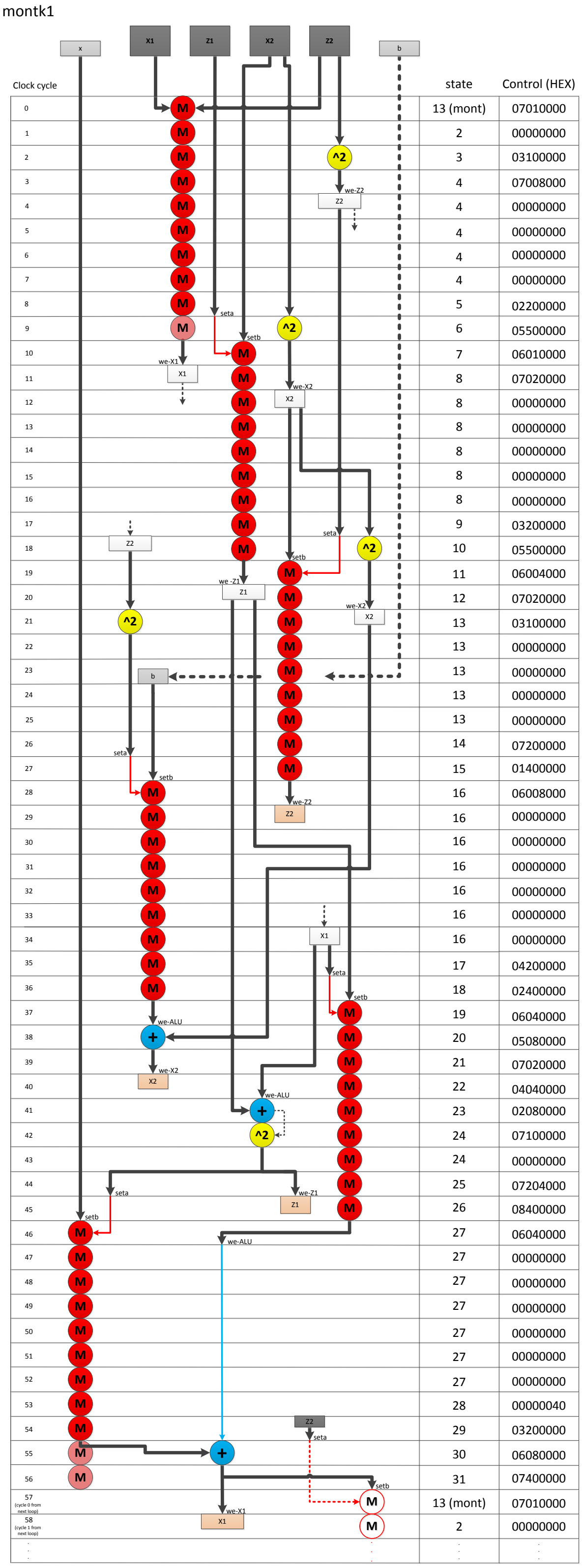
- **Bits 5 to 1 and bit 0:** These bits are sent to two ALU inputs, `be32` and `we32` respectively. When `we32` has the value '1', a 32 bit long value, taken from the data input `r_in32` of the ALU can be saved as part of the 233-bit long internal register of the ALU, overwriting thus 32 bits in this register. `be32` is a 5-bit long word and its value specifies which 32 bits of the internal register should be overwritten by the value taken from `r_in32`.
  The value of bit 0 is set to '1' only once in the initialization and the internal value of the ALU is set to 1.

4.3.3 Montgomery kP-Algorithm Implementation

The Montgomery kP-algorithm (see Algorithm 3 on p. 12) is a balanced method for performing the kP-operation that does not demand the use of dummy operations. As mentioned in section 2.3, every main loop iteration of the algorithm is performed with the same pattern and all results obtained during one loop are used as input parameters for further iterations. The implementation of the Montgomery kP-algorithm is considered to be a strong countermeasure to protect cryptographic implementations at least from SPA attacks [37].

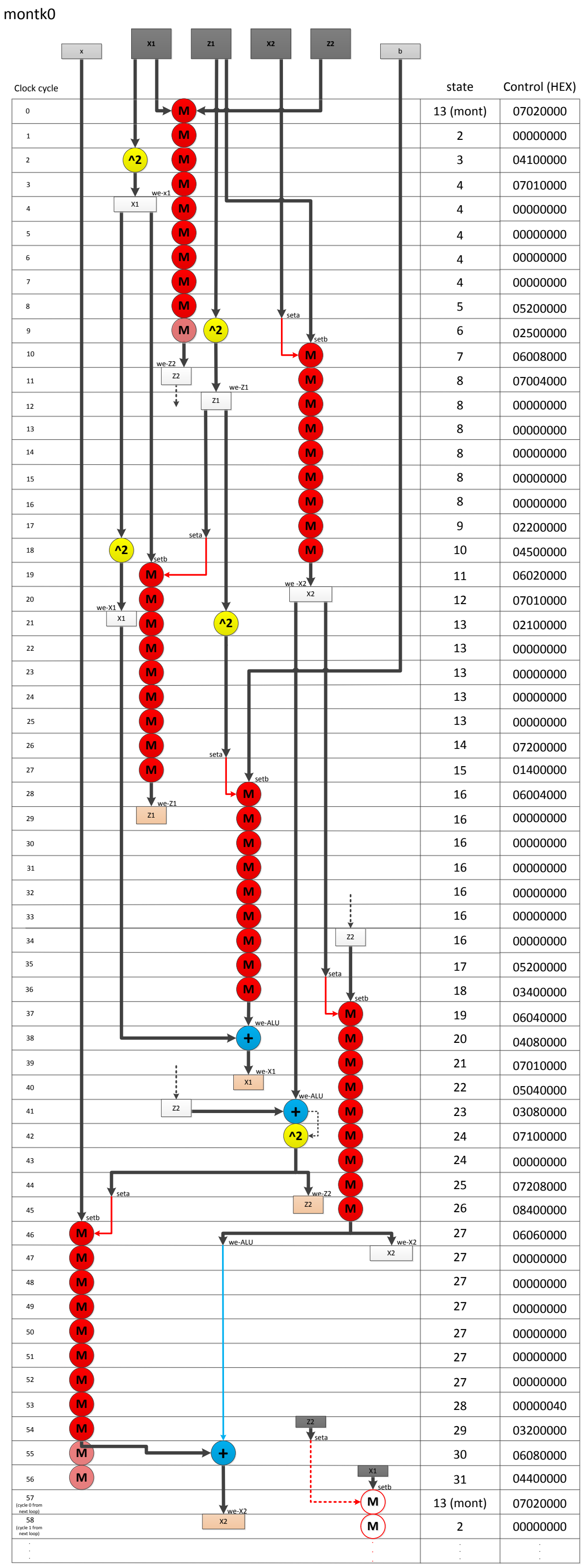
The biggest amount of time needed for the performance of the kP-operation with this design is spent in the inner loop of the Montgomery kP-algorithm. The number of times this loop is performed depends on the value, or rather the length, of the key k [34]. Being dependable from k, these processes play a crucial role in the performance of a side-channel attack on this implementation. This section describes the IHP implementation of the Montgomery algorithm for the kP-operation — that is, the sequence of the operations performed in the 2 possible cases for the inner loop.

The Montgomery kP-algorithm consists of 3 steps: the initialization, the inner loops (for the cases $k_i = 0$ and the $k_i = 1$), and the finalization (these steps are also described in page 39). These 3 steps are implemented in the VHDL file `controller_9_FPGA.vhd` under the names `mont`, `montk1` (for $k_i = 1$), `montk0` (for $k_i = 0$), and `montpost` respectively. Two variables — `state` and `control` — organize the calculation of kP. The variable `state` is responsible for the running step of the Montgomery kP-algorithm, i.e. it can start the programs for `mont`, `montk1`, `montk0` or `montpost`, and it sets the value of the variable `control`. The value of `control` controls the data exchange between the FUs of the ECC design. Figures 4.11 and 4.12 present two flow chart diagrams titled **montk1** and **montk0** respectively. These charts describe the operation flow for both cases of the inner loop.



Clock cycle	state	Control (HEX)
0	13 (mont)	07010000
1	2	00000000
2	3	03100000
3	4	07008000
4	4	00000000
5	4	00000000
6	4	00000000
7	4	00000000
8	5	02200000
9	6	05500000
10	7	06010000
11	8	07020000
12	8	00000000
13	8	00000000
14	8	00000000
15	8	00000000
16	8	00000000
17	9	03200000
18	10	05500000
19	11	06004000
20	12	07020000
21	13	03100000
22	13	00000000
23	13	00000000
24	13	00000000
25	13	00000000
26	14	07200000
27	15	01400000
28	16	06008000
29	16	00000000
30	16	00000000
31	16	00000000
32	16	00000000
33	16	00000000
34	16	00000000
35	17	04200000
36	18	02400000
37	19	06040000
38	20	05080000
39	21	07020000
40	22	04040000
41	23	02080000
42	24	07100000
43	24	00000000
44	25	07204000
45	26	08400000
46	27	06040000
47	27	00000000
48	27	00000000
49	27	00000000
50	27	00000000
51	27	00000000
52	27	00000000
53	28	00000040
54	29	03200000
55	30	06080000
56	31	07400000
57 (cycle 0 from next loop)	13 (mont)	07010000
58 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

Figure 4.11: Flow chart diagram for the program `montk1`.



Clock cycle	state	Control (HEX)
0	13 (mont)	07020000
1	2	00000000
2	3	04100000
3	4	07010000
4	4	00000000
5	4	00000000
6	4	00000000
7	4	00000000
8	5	05200000
9	6	02500000
10	7	06008000
11	8	07004000
12	8	00000000
13	8	00000000
14	8	00000000
15	8	00000000
16	8	00000000
17	9	02200000
18	10	04500000
19	11	06020000
20	12	07010000
21	13	02100000
22	13	00000000
23	13	00000000
24	13	00000000
25	13	00000000
26	14	07200000
27	15	01400000
28	16	06004000
29	16	00000000
30	16	00000000
31	16	00000000
32	16	00000000
33	16	00000000
34	16	00000000
35	17	05200000
36	18	03400000
37	19	06040000
38	20	04080000
39	21	07010000
40	22	05040000
41	23	03080000
42	24	07100000
43	24	00000000
44	25	07208000
45	26	08400000
46	27	06060000
47	27	00000000
48	27	00000000
49	27	00000000
50	27	00000000
51	27	00000000
52	27	00000000
53	28	00000040
54	29	03200000
55	30	06080000
56	31	04400000
57 (cycle 0 from next loop)	13 (mont)	07020000
58 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

Figure 4.12: Flow chart diagram for the program `montk0`.

Below is a short description of the structure of these diagrams:

- On the left side of the diagrams the clock cycles are listed. Each loop consists of 57 cycles.
- The column named 'state' lists the states of the variable program during the loop. Each flow chart describes the complete process operated by one program, either `montk0` or `montk1`. The state described in the cycle 0 of each chart (state = 13 (mont) for every case) is part of the program `mont`. `mont` is the initialization program and it is always entered in its state = 13 for the first cycle of each loop. During this clock cycle, the first multiplication process for a loop starts and the Controller also decides, if a loop for a bit '1' or for a bit '0' should be entered: if the Controller's input signal `is_set` has the value '1', state = 2 of `montk1` is entered in the cycle 1 of the loop. If `is_set` has the value '0', state = 2 of `montk0` is entered in the cycle 1 of the loop.
- On the right side of the flow chart the values of the Controller's `cntr` signal are listed. They are given as hexadecimal numbers, as described in the VHDL file.
- The actions denoted as "`we-`" represent a *word enable* operation performed by the indicated register.
- The red units with the label 'M' represent the activity of the Multiplier. Units displayed with a solid red color represent the current activity of the Multiplier. Units displayed with a transparent red color (cycles 9, 55 and 56 for both programs) represent a non-operating state of the Multiplier[7]. The result obtained from the last operation is saved and can be read. Each loop contains 6 multiplications.
  The non filled units with the label 'M', placed in the cycles 57 and 58 of each loop represent multiplications which belong to the following loop.
- The yellow units with the label '^2' represent the squaring operation of an element of $GF(2^{233})$ performed by the ALU. Each loop contains 5 squaring operations. Each squaring operation needs one clock cycle.
- The blue units with the label '+' represent the addition of two elements of $GF(2^{233})$ performed by the ALU. This operation needs two clock cycles for the calculation of the sum: in the first cycle the first operand is saved in the inner register of the ALU

[7] In the non-operating state of the Multiplier, also named *inactive* state, the new multiplicands for the following multiplication are not available yet and the Multiplier waits for them. The Multiplier has two internal registers named `operand_a` and `operand_b`, where the operands of its partial multiplications are saved. If the Multiplier is in its non-operating state, `operand_a` and `operand_b` are both overwritten with the value '0'.

(when its *word enable* signal is set to '1'). In the second clock cycle the second operand is directly added to the first if its *xor enable* signal is set to '1'. There is a total of 3 additions performed in each loop.

- All registers X1, Z1, X2, Z2 are shown in grey colour when they are being used as input parameters on the beginning of the loop. They are white coloured when being used for saving intermediate values and beige coloured when they store the final values (output values) of the loop.

Both programs, `montk1` and `montk0`, have been implemented the same way. In the Montgomery kP-algorithm, both loops require exact the same number and the same sequence of operations to be executed. They differ only in the use of the registers (X1 and X2; Z1 and Z2) as input and output parameters depending on the value of the processed key bit [18]. In the IHP implementation `montk0` and `montk1` have the following differences, observable in their flow chart diagrams in Figures 4.11 and 4.12 respectively.

1. In cycle 46 of `montk0` one `we-` operation is performed by the ALU and another one by the register X2. This means that 2 `we-` operations are performed on this cycle for `montk0`. In cycle 46 of `montk1` in contrast, only one `we-` operation is performed: `we-`ALU. For this reason, the power consumption during clock cycle 46 is always higher when the program `montk0` is executed than when `montk1` is executed[8].

2. In cycle 44 for both programs a `seta` operation is performed. The output value of the ALU is saved in an internal register of the Multiplier, named `polynomial_a`. `polynomial_a` is overwritten every time a `seta` operation is performed. Also during cycle 44, a `we-` operation is executed and the output value of the ALU is saved either in register Z1 (for `montk1`) or in register Z2 (for `montk0`). In cycle 54, a `seta` operation is again performed, which means that the value of `polynomial_a` is again overwritten. Both programs, `montk1` and `montk0`, use the value of register Z2 as input to perform the `seta` operation in cycle 54. For `montk1` the value of Z2 has been set for the last time during cycle 28, so when `seta` is executed in cycle 54, `polynomial_a` is overwritten with a complete new value. For `montk0` on the other hand, the value of Z2 has been set for the last time during cycle 44, using the same input that was used for performing the `seta` operation. This means that `polynomial_a` receives the same input

[8] The performance of the `we-`X2-operation is not needed since the value of the register X2 is overwritten again in the clock cycle 0 of the following loop, being this the last operation performed by `montk0`. Between the clock cycles 46 and 57 (clock cycle 0 for the next loop), the value of X2 is never read. It is believed that the performed `we-`X2-operation has been implemented by mistake.

data that it already has and the value of the register is not overwritten but it is just kept.

3. In cycle 56, the last clock cycle in the loop, a `setb` operation is performed for setting the second multiplicand for the multiplication for the next loop. In `montk0` the value of the second multiplicand is obtained from register X1, in `montk1` this value is obtained from the ALU.

The flow charts shown in the previous section describe the inner loops of the Montgomery kP-algorithm by depicting every operation being executed in each clock cycle. Figures in Appendix B show the complete flow diagram for the kP-operation with the use of a small key $k = 2cc$. To illustrate the leakage sources, the power consumption is shown in the same diagram as traces for each FU of ECC designs.

4.4 Observations

The analysis of the diagrams in Figures 4.11 and 4.12 has been a key part for the realisation of the security assessment described in this chapter. This section lists the causes of information leakage that were detected through comparison the power consumption of the `montk1` and `montk0` programs. These leakage sources are explained based on observations made on the implementation's characteristics that have been listed in the previous sections and in the simulated PTs of the FUs of the ECC design.

1) Easy identification of the boundaries between the loops

Periodically repeated forms of the PT simplify the detection of how long (how many clock cycles) each key bit needs to be processed. PTs for the analysed design show two of such characteristic forms: one being a "plateau" in the curve and the other one being a significant dip. Both of these characteristics appear right after each other in all slots for each bit processing, also for each loop. Figure 4.13 shows these characteristics of the PT.

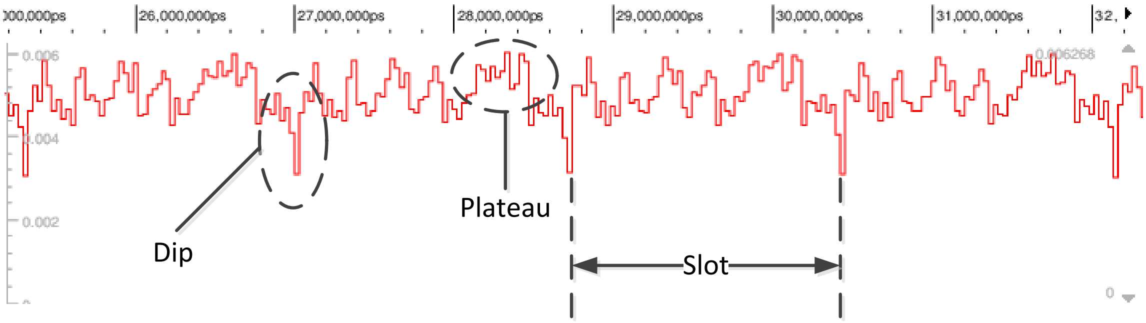


Figure 4.13: Part of a PT of the ECC design during the performance of the kP-operation. The graph shows power (in W) over a period of time (in ps). This part of a PT corresponds to the processing of 4 bits of a 10 bit long key. The dips in the trace's curve are caused due to a very low power consumption of the design during only one clock cycle. A few clock cycles before each dip, a plateau can be observed. The space between 2 dips corresponds to one slot and is 57 clock cycles long.

The following conclusions have been made related to both of these anomalies through analysing the flow charts for the inner loops of the Montgomery kP-algorithm:

- **Plateau:** This object is seen between the clock cycles 38 and 42 of each slot (5 cycles in total). 3 operations are performed by the ALU during this time. Two of them are additions and one is a squaring operation. This relative big number of ALU operations is only performed in this short period of time between these clock cycles and it demands constant activity from the ALU during this period, thus leading to a continuous high power consumption of this FU. Such a number of ALU operations in a similar short time interval is not performed in any other period during the loop.

- **Dip:** This anomaly is seen at the end of each loop. The reason for the dip is the fact that, having the highest energy consumption of all FUs, the Multiplier is not active during the last two clock cycles (55 and 56). After cycle 54, the last multiplication in the loop has been completed and its result can be read. The Multiplier stays in its non active state for the following two clock cycles because it needs to wait for its next input parameters. The Multiplier's energy consumption reaches a value close to zero during cycle 56 when none of its gates are switched.

This implies a big decrease of the energy consumption of the entire design at this clock cycle.

2) Correct key extractions when the DPA is performed on individual FUs

Figure 4.8 (see p. 46) shows the relative correctness of the key extractions obtained when using PTs of the individual blocks of the ECC design. More than 10 out of a total of 57 key candidates are extracted with a correctness of 100% when using PTs of the registers (see Fig 4.8 b) and c)).

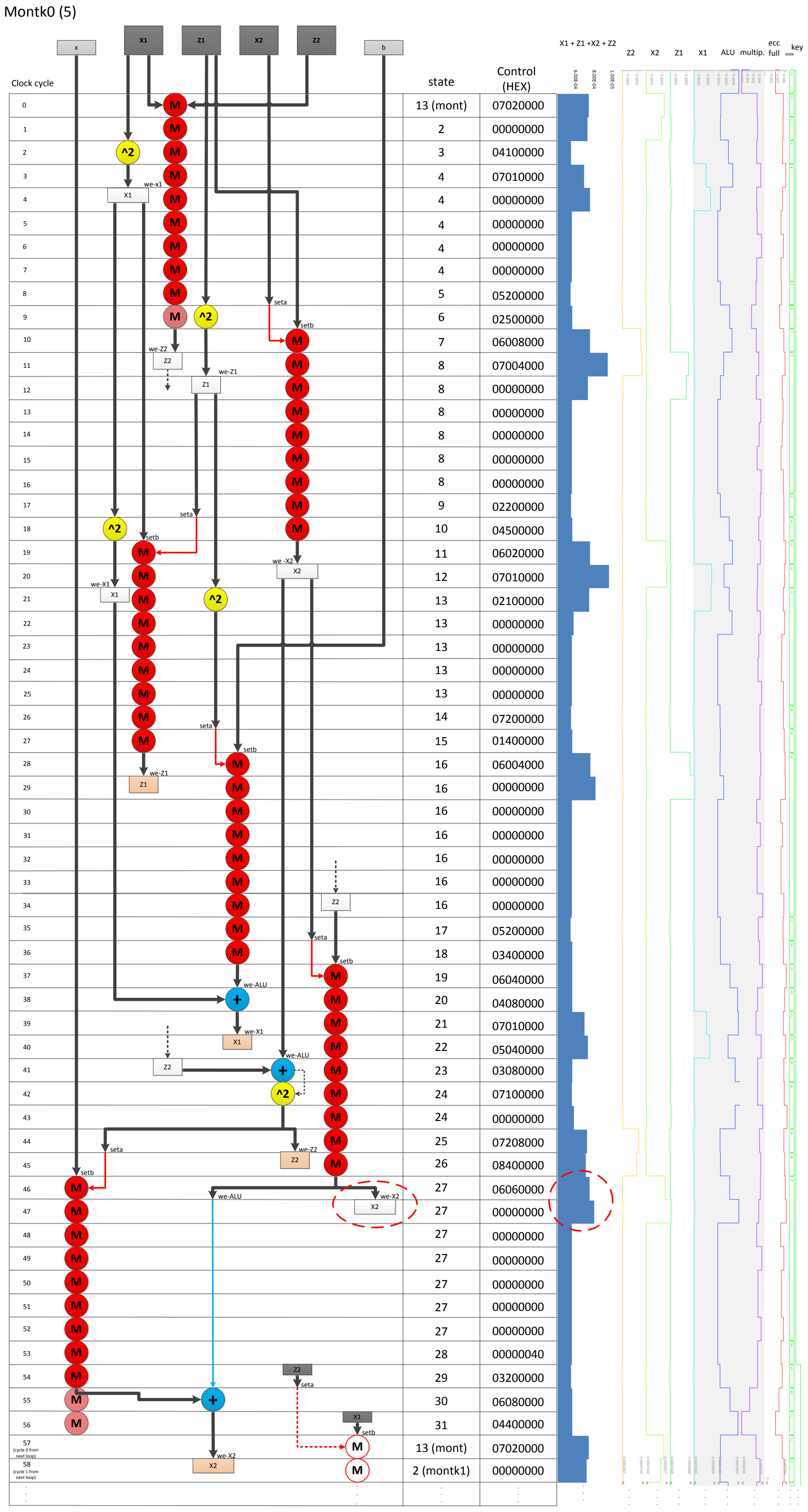
Some of these correct extractions are caused because the registers are used differently in `montk1` and `montk0`. For example, during a clock cycle in `montk1` the register X1 is overwritten and in `montk0` instead, the register X2 is overwritten in the corresponding clock cycle. The registers consume power dynamically whenever their values are overwritten.
Other correct key extractions are done for clock cycles on which the output value of an FU is written to the Bus and used as input for another FU in the design. For example, during a clock cycle in `montk1` the value of the register X1 is written to the Bus; in `montk0` instead, the value of X2 is written to the Bus in the corresponding clock cycle. Whenever data is written to the Bus, all FUs receive this data on their data inputs, even though only one of them actually uses it (see section 4.3.1). Every time the data input of an FU is changed, the FU consumes a small amount of *internal* power[9]. The internal power consumption of a register is significantly different when its data input and output are the same as when its data input and output have different values.

[9] The internal power is the power consumed by an FU when its input is changed but its output is not changed. Through the change on its input, a small number of transistors of the FU's gates are switching [36].

3) Unbalanced implementation of the ECC design

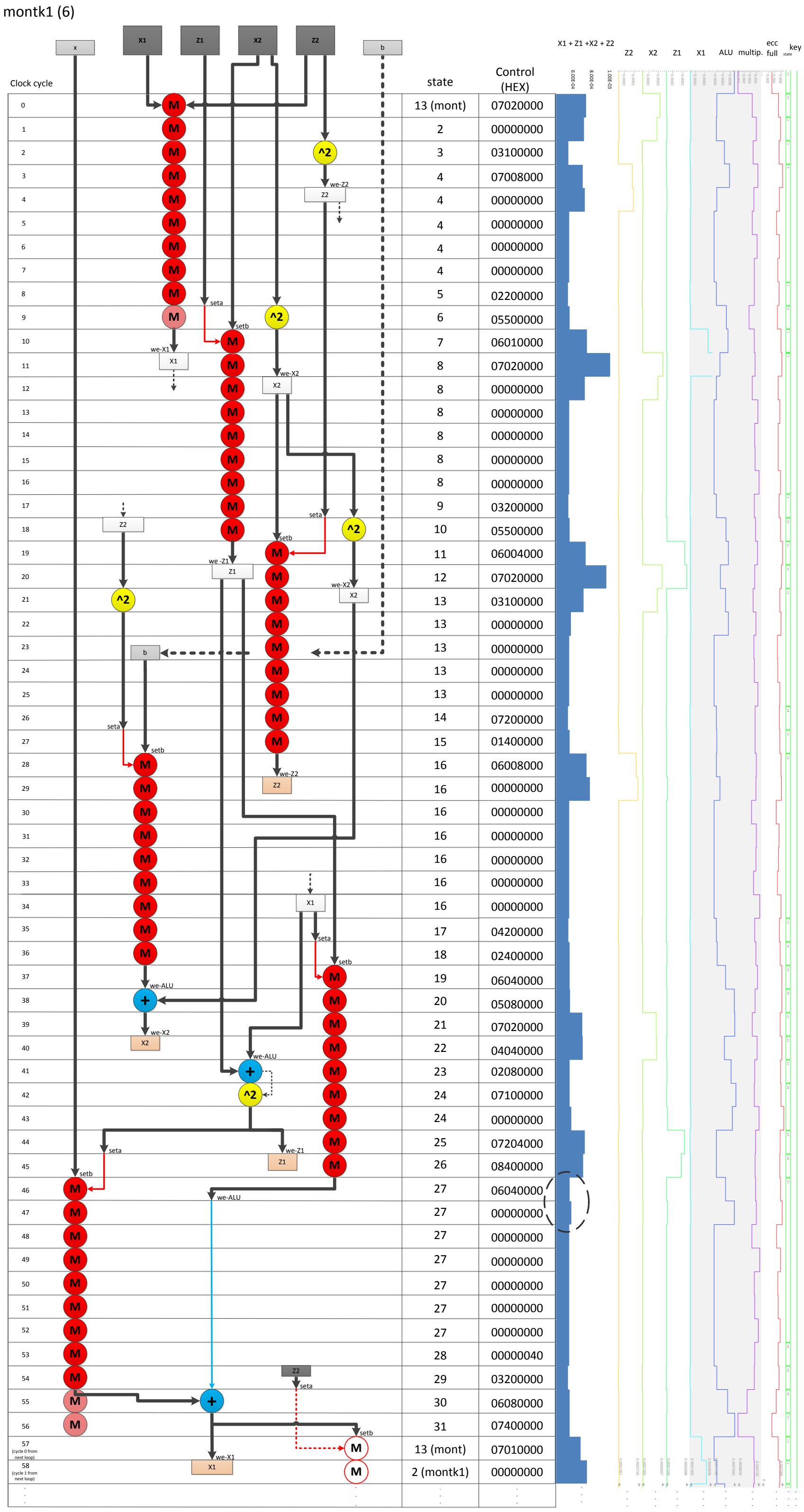
The loop executions in the Montgomery kP-algorithm depend on the value of the processed key bit. The details about the algorithm implementation, explained in section 4.3.3, point out that there are 3 differences in the programs `montk0` and `montk1`. The diagrams **montk0 (5)** in Figure 4.14 and **montk1 (6)** in Figure 4.15 show the operations being executed for each loop and the power consumed by these operations. The power consumption is shown as part of a PT on the right side of the diagrams. This illustrates the effects of the unbalance in the implementation of the Montgomery kP-algorithm[10]. These are PTs simulated for the complete ECC design and for its individual FUs. Below is a comparison of the operations and the power consumptions for programs `montk1` and `montk0` during the clock cycles that were pointed out as leakage sources in section 4.3.3.

[10] The diagrams displayed in Figures 4.14 and 4.15 are part of the diagrams included in Appendix B, which show the complete flow diagram for the kP-operation when a small key is used.



Clock cycle	state	Control (HEX)
0	13 (mont)	07020000
1	2	00000000
2	3	04100000
3	4	07010000
4	4	00000000
5	4	00000000
6	4	00000000
7	4	00000000
8	5	05200000
9	6	02500000
10	7	06008000
11	8	07004000
12	8	00000000
13	8	00000000
14	8	00000000
15	8	00000000
16	8	00000000
17	9	02200000
18	10	04500000
19	11	06020000
20	12	07010000
21	13	02100000
22	13	00000000
23	13	00000000
24	13	00000000
25	13	00000000
26	14	07200000
27	15	01400000
28	16	06004000
29	16	00000000
30	16	00000000
31	16	00000000
32	16	00000000
33	16	00000000
34	16	00000000
35	17	05200000
36	18	03400000
37	19	06040000
38	20	04080000
39	21	07010000
40	22	05040000
41	23	03080000
42	24	07100000
43	24	00000000
44	25	07208000
45	26	08400000
46	27	06060000
47	27	00000000
48	27	00000000
49	27	00000000
50	27	00000000
51	27	00000000
52	27	00000000
53	28	00000040
54	29	03200000
55	30	06080000
56	31	04400000
57 (cycle 0 from next loop)	13 (mont)	07020000
58 (cycle 1 from next loop)	2 (montk1)	00000000
. . .	. . .	. . .

Figure 4.14: Flow chart diagram for `montk0` with PTs simulated during its execution.



Clock cycle	state	Control (HEX)
0	13 (mont)	07020000
1	2	00000000
2	3	03100000
3	4	07008000
4	4	00000000
5	4	00000000
6	4	00000000
7	4	00000000
8	5	02200000
9	6	05500000
10	7	06010000
11	8	07020000
12	8	00000000
13	8	00000000
14	8	00000000
15	8	00000000
16	8	00000000
17	9	03200000
18	10	05500000
19	11	06004000
20	12	07020000
21	13	03100000
22	13	00000000
23	13	00000000
24	13	00000000
25	13	00000000
26	14	07200000
27	15	01400000
28	16	06008000
29	16	00000000
30	16	00000000
31	16	00000000
32	16	00000000
33	16	00000000
34	16	00000000
35	17	04200000
36	18	02400000
37	19	06040000
38	20	05080000
39	21	07020000
40	22	04040000
41	23	02080000
42	24	07100000
43	24	00000000
44	25	07204000
45	26	08400000
46	27	06040000
47	27	00000000
48	27	00000000
49	27	00000000
50	27	00000000
51	27	00000000
52	27	00000000
53	28	00000040
54	29	03200000
55	30	06080000
56	31	07400000
57 (cycle 0 from next loop)	13 (mont)	07010000
58 (cycle 1 from next loop)	2 (montk1)	00000000
⋮	⋮	⋮

Figure 4.15: Flow chart diagram for montk1 with PTs simulated during its execution.

- **Unbalance in cycle 46**

  In `montk0` two `we-` operations are performed during this cycle: one `we-` operation is performed by the ALU and another one by the register X2.

  In `montk1` only one `we-` operation is performed: the one of the ALU.

  This means that in `montk0` the gates of two registers switched, while in `montk1` only the gates of one register switched during this clock cycle. Consequently, the power consumption of the sum of all registers is higher for `montk0` than for `montk1` (compare PT part circled in red in Figure 4.14 and the part circled in black in Figure 4.15). This explains why the difference-of-means test reached a key-extraction with over 90% correctness for this clock cycle (see point 47 in Figures 4.5 and 4.7).

- **Unbalance in cycle 54**

  For the reasons mentioned in section 4.3.3 (see point 2 in p. 57), more gates of the Multiplier switched during this clock cycle when `montk1` is executed than when `montk0` is executed.

  Thus more power is consumed during this and the following clock cycle (55) when `montk1` is executed. The difference-of-means curve shows a relative correctness of the key extraction of 90% for clock cycle 55 (see point 56 in Figures 4.5 and 4.7). The difference-of-means test performed on the Multiplier block also gives this point as a strong SCA leakage source (see point 56 in Figure 4.8 a)).

- **Unbalance in cycle 56**

  The differences in the implementations of `montk0` and `montk1` regarding this cycle are explained detailed in section 4.3.3 (see point 3 on p. 58). Here is a short overview of these differences.

  In both programs, the Multiplier obtains through a `setb` operation its second multiplicand as input during this clock cycle. The difference is that in `montk0` this multiplicand is read from register X1 and in `montk1` from the ALU, i.e. in `montk0` the value of register X1 is written to the Bus and in `montk1` the value of the ALU is written to the Bus. The writing of different values to the Bus is one cause of the information leakage (see observation 2 on p. 60). This type of information leakage becomes significant in this case because the ALU and the register X1 are implemented with flip-flops of different types due to their functionalities[11].

[11] The results of the difference-of-means test for register X1 show that a key is extracted with 100% correctness for clock cycle 56 (see point 57 in Figure 4.8 b)). For the ALU, a key-extraction with 30% correctness could be done for this clock cycle (see point 57 in Figure 4.8 d)).

The internal and dynamic power consumptions of the ALU and the registers are significantly different. This is the reason why this unbalance in the implementation — the writing to the Bus of the value of a register in `montk0` and the writing to the Bus of the value of the ALU in `montk1` — is more observable compared to the case when in both programs values of two different registers are written to the Bus.

This difference is small and not easily observable in the power consumption of the complete ECC design if the Multiplier is active. The Multiplier consumes the biggest amount of power in this design. If the Multiplier is active, its power consumption *covers* the activities of other FUs.
It can be seen in the diagrams **montk0 (5)** and **montk1 (6)** in Figures 4.14 and 4.15 respectively, that by clock cycle 56 the power consumption of the Multiplier reaches a value near zero. No gates are switched in the Multiplier. The significantly reduced activity of the Multiplier reduces its *covering effect*. The results of the difference-of-means test for the complete ECC design show that a key-extraction with a correctness of almost 90% can be done for cycle 56 (see point 57 in Figure 4.7).

- **Unbalance in cycle 57 (also represented as cycle 0 of the following loop)**
  In `montk0` during clock cycles 56 and 57 two different values are written to the Bus: the value of register X1 in cycle 56 and the value of the ALU in cycle 57.
  In `montk1` the same value of the ALU is written to the Bus during both cycles (56 and 57).
  The FUs do not demand internal power consumption during cycle 57 when `montk1` has been executed since their data inputs are not changed. Thus, the power consumption of all FUs is higher during clock cycle 57 (0) when `montk0` has been executed as when `montk1` has been executed. The activity in the Bus for both programs during these clock cycles is summarized in Table 4.2.

Table 4.2: Values written to the Bus during cycles 56 and 57 (0).

	cycle 56	**cycle 57 (0)**
`montk0`	value of X1	value of ALU
`montk1`	value of ALU	value of ALU

The different power consumptions during clock cycle 0 for the programs montk0 and montk1 can be seen in Figure 4.16.

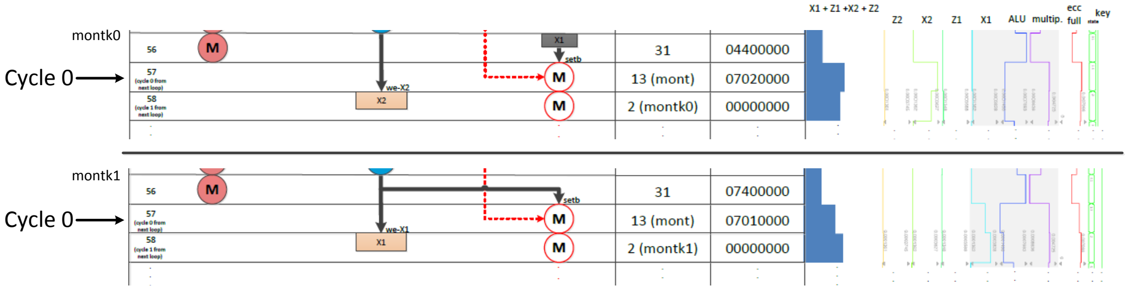


Figure 4.16: Comparison of the power consumption of all registers during the first clock cycle of a loop (see cycle 0).
The upper part of the diagram corresponds to montk0 and the lower part to montk1.

This key bit value dependant power consumption is only partially covered by the Multiplier. The Multiplier is active for the first time after having been inactive for 2 clock cycles. This means that in the previous clock cycles, cycles 55 and 56, its internal registers operand_a and operand_b and other gates of its partial multiplier have the value '0'. In clock cycle 0, the Multiplier performs a partial multiplication again and its internal registers are overwritten with the values of the multiplicands. The amount of power consumed to calculate the partial product is smaller after such "zero"-initialization than it is in the usual cases, when the initialized values of all registers is not '0'[12].

This means that the Multiplier always consumes a smaller amount of power during clock cycle 0 than it usually[13] does. The complete ECC design's power consumption is consequently smaller during this clock cycle and the details in the power consumptions of individual FUs are more observable: the key was extracted with a relative correctness of 100%.

It can be concluded that the interchangeable use of the registers in programs montk1 and montk0 as input parameters for other FUs leads to information leakage. A possible

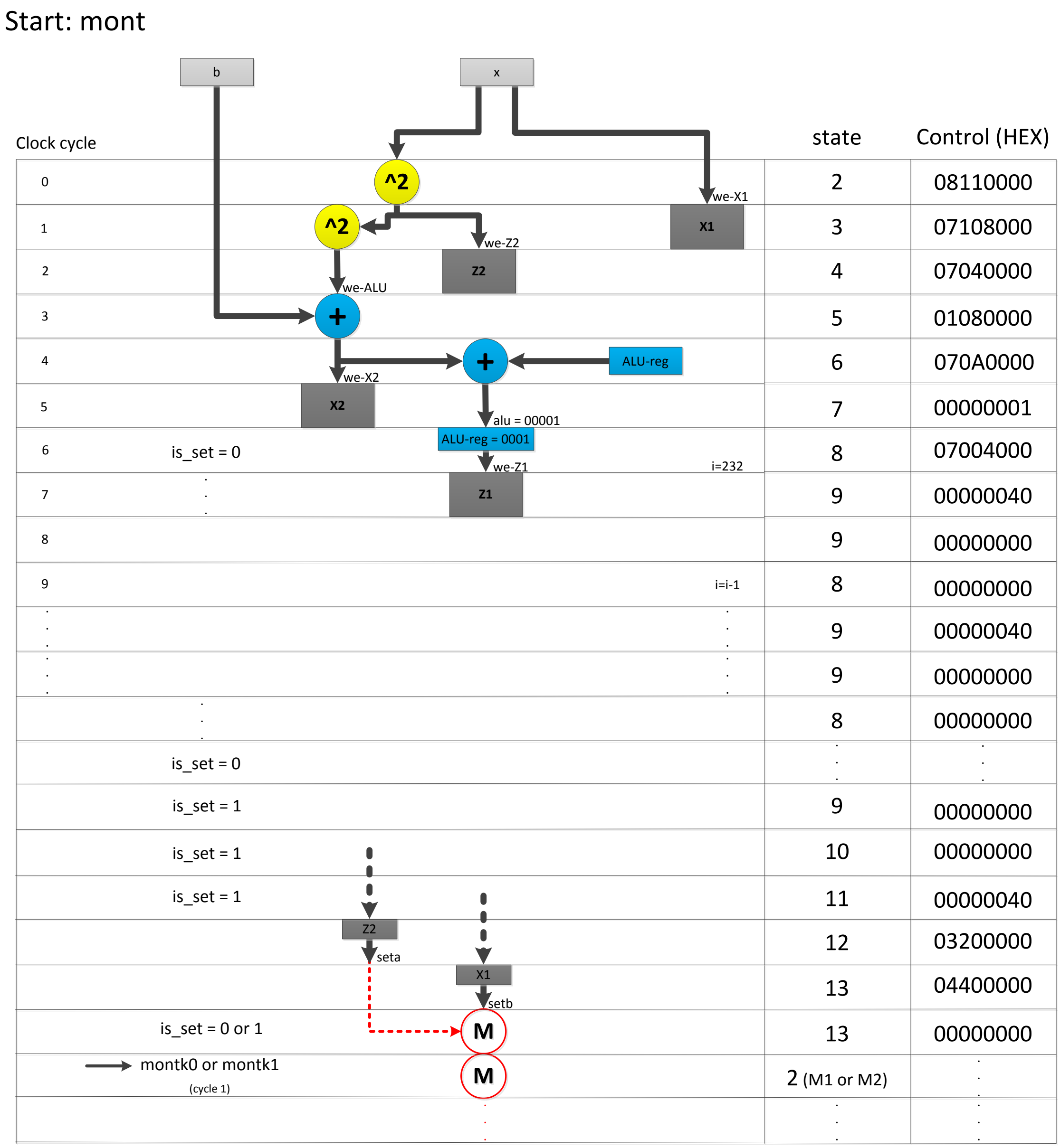
[12] If all flip-flops inside the register contain the value 0, only some of them have to be switched in order to set the new value in the register. If the register holds a value different to 0, a bigger amount of flip flops have to be switched (from 0 to 1 and from 1 to 0) to set the new value. The flip-flops used for the implementation of these internal registers consume less power when they are switched from 0 to 1 as when they are switched from 1 to 0 [10].

[13] i.e. in comparison to clock cycles 1 – 54.

countermeasure against this would be the re-design of the system architecture of this implementation: the internal power consumption of each FU would not be changed so frequently if they would not receive a different data value from the Bus every clock cycle. Nevertheless, the leakage through internal power consumption of the FUs becomes significant for this design's vulnerability against SCA only if the Multiplier consumes less power as it usually does, i.e. if its covering effect is not present.

4) The length of a PT depends on the size of the key

By using a key of small length, the PT of the kP-operation has a short size. This could let an attacker know about the size of the key, providing him information about the implemented algorithm or letting him know that the performance of a brute force attack is possible in case of a key with a short length. Figure 4.17 displays the entire initialization process for the kP-operation. Some activities can be seen on the beginning and then, depending on the length of the key, no operations are performed for a certain time. Operations are performed again as soon as the processing for the first key bit starts.



Clock cycle		state	Control (HEX)
0		2	08110000
1		3	07108000
2		4	07040000
3		5	01080000
4		6	070A0000
5		7	00000001
6	is_set = 0	8	07004000
7		9	00000040
8		9	00000000
9	i=i-1	8	00000000
.	.	9	00000040
.	.	9	00000000
	.	8	00000000
	is_set = 0	.	.
	is_set = 1	9	00000000
	is_set = 1	10	00000000
	is_set = 1	11	00000040
		12	03200000
		13	04400000
	is_set = 0 or 1	13	00000000
	→ montk0 or montk1 (cycle 1)	2 (M1 or M2)	.
	.	.	.

Figure 4.17: Flow chart of the initialization part of the Montgomery kP-algorithm.

5) The first iteration of the main loop reveals information about the corresponding key bit value

The PTs for the execution of the first loop iteration, i.e. for the processing of key bit k_{l-2} look different compared to those for the execution of the rest of the loops. Figure 4.18 shows simulated PTs for two different keys. The keys have the same length and the MSB of both is $k_{l-1} = 1$. The key bit k_{l-2} is different for each case: for the first key, $k_{l-2} = 0$ and for the second key $k_{l-2} = 1$. The first slot in the PTs corresponds to the first loop iteration. For the upper curve $k_{l-2} = 0$ and for the lower curve $k_{l-2} = 1$.

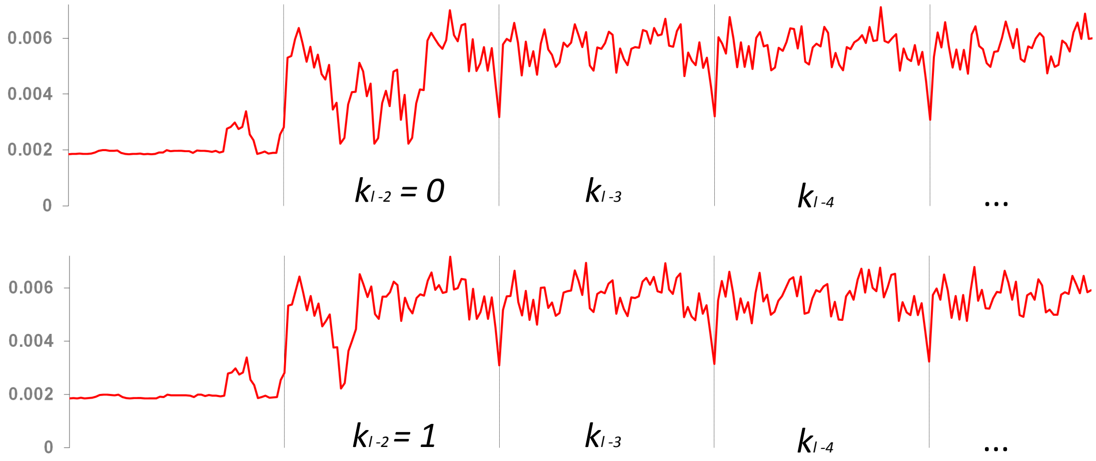


Figure 4.18: Simulated PTs for keys with the same MSB value and different value for k_{l-2}. For the upper curve, the power consumption along the slot is lower and more irregular than for the curve below.

In the diagram of **Start: mont**, displayed in Figure 4.17, it can be observed that the register Z1 is initialized with the value '1'. This means that the first iteration of the main loop, i.e. the iteration for the key bit k_{l-2} will take the register Z1 with this value as input. Any multiplication performed with Z1 $= 1$ as operand will result in the value of the other operand. The same applies for any multiplication performed with the result of a squaring operation that has taken Z1 $= 1$ as input, since the result to this squaring operation is also 1. The performance of such operations does not demand a big amount of power consumption since the values in internal registers in these cases are not overwritten. For this reason, the PTs measured during the execution of the first loop iteration look different as those measured during the rest of the loop iterations. `montk0` and `montk1` use the register Z1 differently as input. Thus, a different amount of multiplications with an operand of value 1 is performed when the first loop iteration takes place for the key bit $k_{l-2} = 0$ as for the key bit $k_{l-2} = 1$. `montk0` performs a total of 3 multiplication with an operand of value 1, while `montk1` performs only 1 of such multiplications. This means that the power consumption is higher in the first slot if $k_{l-2} = 1$ and `monkt1` is executed. In this context the bit value of k_{l-2} can be easily

identified.
Revealing the value of one key bit so easily may have big negative consequences. By knowing for sure which key bit is being used during a specific slot, it is possible to use this slot as a template for attacks based on FI, for example in the FSA attack (see section 3.2.2).

The observations mentioned in this chapter helped to identify the reasons that caused this ECC implementation's vulnerabilities to SCA attacks. The design's functionality and its power consumption have been displayed in diagrams. This, alongside the difference-of-means test results, help to understand which of the operations are strong leakage sources for an SCA. By taking these properties into consideration when re-designing this crypto-algorithm, a more secure implementations can be developed.

5 Re-design of the IHP ECC Implementation

The observations made through the security assessment described in the previous chapter helped identify the causes of a successful performance of a DPA attack using the difference-of-means test on the IHP ECC implementation. The SCA leakage sources were identified. By understanding the implementation's architecture and functionalities, it is possible to re-design it in such a way, that these leakage sources are no longer present and that the implementation's vulnerability to SCA attacks decreases.

This chapter describes the adjustments made on the ECC design, which improved its resistance against SCA. Several changes have been done in the system's architecture and thereby, special effort was made so that the increase on the system's area and energy consumption caused by the implementation of countermeasures remained as small as possible. The new design's architecture, its improved efficiency and security features are displayed in diagrams the same way as they are displayed in the previous chapter.

Section 5.1 describes the changes made on the system's architecture, the configuration of the signal `cntr`, and how these helped to re-design the operation sequences in the Montgomery kP-algorithm. Results regarding the chip area and energy consumption of the re-designed implementation are compared to the chip area and energy consumption of the original IHP ECC design in section 5.2. Two PA attacks are performed on the re-designed implementation: a DPA attack using the difference-of-means test (see section 5.3) and a ComPA attack with PTs measured from the execution of the re-designed implementation on an FPGA (see section 5.4). The results of both tests are presented to show how the ECC implementation's vulnerability to such attacks changed after its re-design.

5.1 Technical Changes

To achieve resistance against a DPA attack using the difference-of-means test, the operation sequence of the implemented Montgomery kP-algorithm (see section 4.3.3) was modified. The sequence of the single operations in the algorithm's main loop and the regularity of each operation's execution were changed. Both programs, `montk1` and `montk0` were implemented in the same way, but without the disadvantages mentioned in section 4.3.3 (see p. 57) and thus, a balanced algorithm implementation was achieved.

For the re-design of the Montgomery kP-algorithm, two new registers — X3 and X4 — are additionally used for saving values during the algorithm's main loop executions. As a consequence, two further bits of the signal `cntr` are used for the communication with these registers. The resulting state of the ECC's system architecture, the extended configuration of `cntr` and the re-designed operation sequence of the main loops of the kP-algorithm are described in this section.

5.1.1 System Architecture

The block diagram in Figure 5.1 shows all entities that are part on the re-designed execution of the kP-operation and how they are connected with each other. It consists of the same entities as described in 4.3.1, two additional registers X3 and X4, and the corresponding single bits of the signal `cntr` for the communication with these two registers.

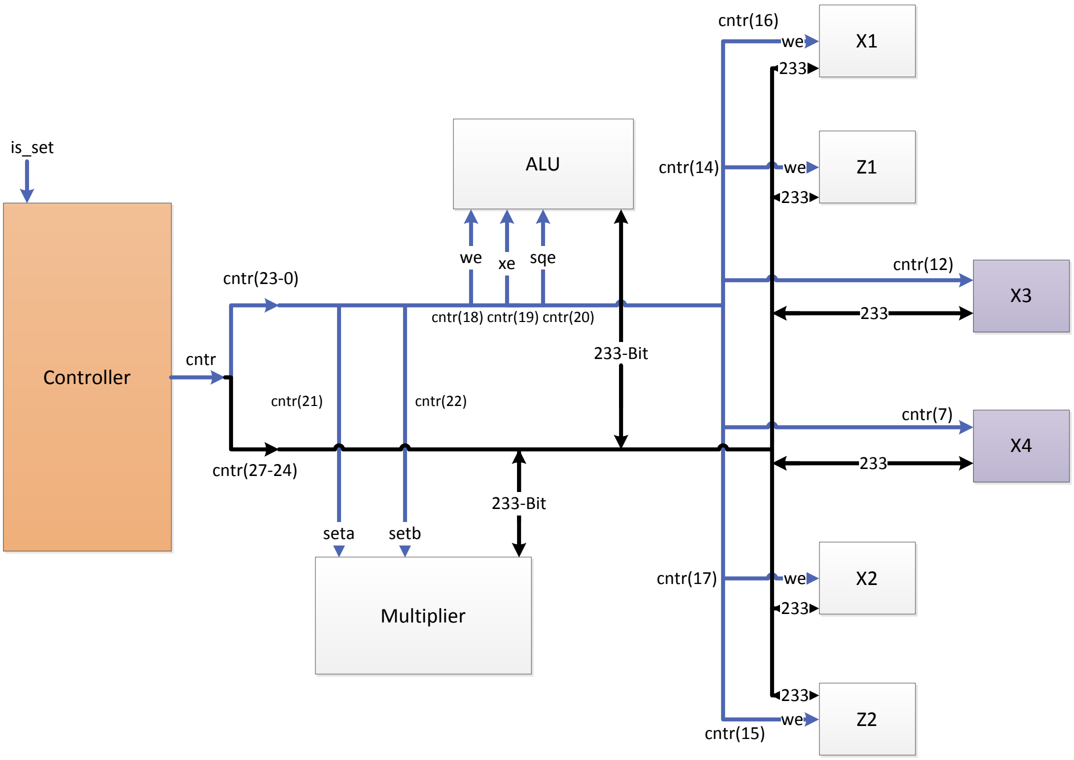


Figure 5.1: New structure of the IHP ECC design.
The entities shown in this diagram execute the kP-operation and are connected with each other through a bus channel. The FUs Multiplier and ALU perform mathematical operations in $GF(2^{233})$. The FUs X1, Z1, X2, Z2, X3 and X4 are 233-bit long registers. Registers X3 and X4 are needed for a balanced operation sequence and a time optimized execution of the kP-operation.

Like the other external registers, X3 and X4 are 233-bit long storage registers. The data from the Bus is saved in the registers every time their input signal '`we`' (word enable) has the value '1'. These registers are used to save intermediate values during the execution of the Montgomery kP-algorithm's main loops.

All other entities work the same way as described in 4.3.1.

5.1.2 Control Signal

The Controller's output signal `cntr` controls the Bus access for all other FUs . The bits 27 to 24 form a bit word, which is received by the Bus as the input signal `sel`. The value of signal `sel` determines which FU's data output should be written to the Bus.

With the two additional FUs, registers X3 and X4, two single bits from `cntr` needed to be assigned as their `we` input signals. Figure 5.2 illustrates the configuration of the `cntr` signal after these new bit assignments. In the original IHP ECC design, 8 bits of `cntr` are not used by the Controller (see section 4.3.2). Two of these bits, bit 7 and bit 12, are now assigned as `we` inputs of the registers X4 and X3 respectively.

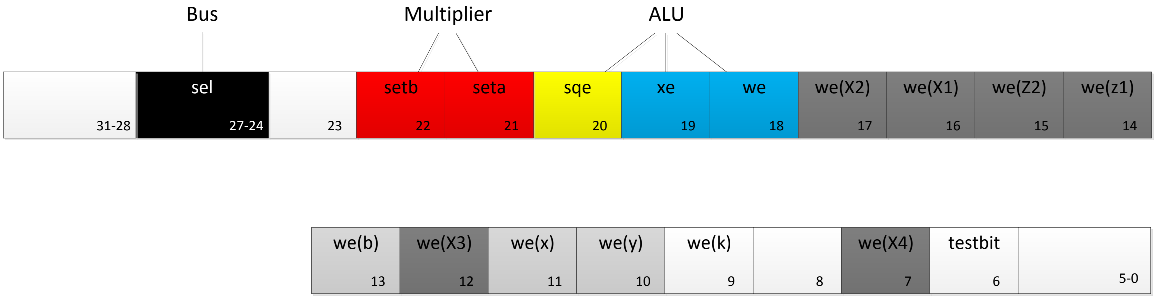


Figure 5.2: New configuration of the signal `cntr`.

To control the Bus access to the output data of the new registers, the Bus needed to recognize two further possible values of its input signal `sel`. The binary representation of the numbers “10” and “11”, formed by the bits 27 to 24 of `cntr`, were assigned for this. This way, the signal `sel` could have two further possible input values for the Bus. Table 5.1 shows how the possible states of `sel` were extended:

Table 5.1: Re-designed implementation: values of the signal `cntr` for the Bus.
The additional values of `sel` for writing the outputs of the new registers are marked.

`sel`, Used Value	Signal Name	Description
0	`o_ext_reg_r_out`	output of external regiser is written to the Bus
1	`o_b_r_out`	output of register b is written to the Bus
2	`o_z1_r_out`	output of register Z1 is written to the Bus
3	`o_z2_r_out`	output of register Z2 is written to the Bus
4	`o_x1_r_out`	output of register X1 is written to the Bus
5	`o_x2_r_out`	output of register X2 is written to the Bus
6	`o_multiply_result`	output of Multiplier is written to the Bus
7	`o_alu_r_out`	output of ALU is written to the Bus
8	`o_x_r_out`	output of register x is written to the Bus
9	`o_y_r_out`	output of register y is written to the Bus
10	`o_x3_r_out`	output of the register X3 is written to the Bus
11	`o_x4_r_out`	output of the register X4 is written to the Bus

The new configuration of the signal `cntr` has the following differences in comparison to its original configuration, (see section 4.3.2): bits 5 to 0 are not used any more as inputs for the ALU (see Figure 5.2). They are no longer needed as it will be explained later in this chapter. These bits remain unused for the entire execution of the kP-operation. The length of `cntr` can be reduced or the unused bits can be used when implementing further functionalities in this ECC design. The following subsection describes the re-designed implementation of the kP-operation.

5.1.3 Re-designed Implementation of the Montgomery kP-Algorithm

As explained in section 4.3.3, the execution time of the kP-operation consists mostly of the time needed for the inner loops of the Montgomery kP-algorithm. Depending on the value of the processed key bit, program `montk1` or program `montk0` is executed.

The main changes made on these programs for their re-design regard the sequence of the single operations performed by them and the regularity of each operation's execution. This way, the re-designed implementation of the algorithm is time optimized: the processing of each key bit only needs 54 clock cycles now, these are 3 cycles less in comparison to the original implementation.

For the implementation of the Montgomery kP-algorithm described in [18], the processing of each key bit requires always the same number and the same sequence of operations. Only the use of the registers (X1 and X2; Z1 and Z2) as input and output parameters for the loop depends on the value of the processed key bit. For example, X1 and Z2 are used to perform the first multiplication of the loop if a key bit with value '1' is being processed. Otherwise, the first multiplication uses X2 and Z1 as inputs. The multiplication, addition and squaring operations are performed one by one at different clock cycles each. This is not time and energy consumption efficient since multiplications need a longer execution time and demand higher power consumptions than the rest of the operations.
The operation sequence in the loops of the original IHP ECC implementation differs from the one introduced in [18]. In the original IHP ECC implementation, the data from registers X1 and Z2 is always used as input for the first multiplication, independent of the value of the key bit being processed. This was done in order to achieve a faster execution time of the loops. Since the execution of the multiplications needed 9 clock cycles and the rest of the operations were performed by different FUs, the operation sequence was implemented in such a way, that other arithmetic operations were performed in parallel to the multiplications of a product.

To achieve a balanced and time optimized implementation, additional instructions were implemented on both programs `montk1` and `montk0` as part of the re-design process. These are conditional instructions with which the programs can be executed either in their normal or in a variant form. During the execution of these programs, the conditional instructions check which program (`montk1` or `montk0`) was previously executed. Depending on this, some arithmetic operations are executed at different clock cycles.

The programs are executed in their variant forms every time after `montk1` has been executed, otherwise the programs are executed in their normal form. Every time `montk1` is executed, an internal variable `variant` is set to the value '1'. Every time `montk0` is executed, `variant` is set to '0'. So the value of `variant` is always checked in order to know which program has been previously executed. Table 5.2 shows an example of the execution sequence of these programs and their variations for the case that the key k = 10101100 is being processed. Hereby (v) is written next to the names of the programs when they are executed in their variant forms.

Table 5.2: Example of the new execution sequence of `montk1` and `montk0` for the processing of $k = 10101100$.

k_i	1	0	1	0	1	1	0	0
`prg`	`mont`	`montk0`	`montk1`	`montk0`(v)	`montk1`	`montk1`(v)	`montk0`(v)	`montk0`

In the re-designed IHP ECC implementation, when `montk0` and `montk1` are executed in their normal form, the first multiplication at the beginning of the program execution is performed using the data from registers X1 and Z2 as input, as in the original implementation. The `seta` and `setb` signals, with which the Multiplier saves its multiplicands to its inner registers, are always set during the last two cycles (52 and 53) of the previous iteration. Concerning these two clock cycles for both programs, an important difference should be mentioned:

- In `montk0` the `seta` and `setb` signals are set to obtain the data from registers Z2 and X1 respectively. These registers already hold their final output value for the loop and they can thus be used as input parameters for the following loop.
- In `montk1` in contrast, at the time the `seta` and `setb` operations are performed, register Z2 has not yet been overwritten with its final output value for the loop and Z2 cannot be used as an input parameter for the first multiplication of the following loop. For this reason, the `seta` and `setb` operations performed in cycles 52 and 53 of `montk1` are performed with the values of registers X2 and Z1 respectively. X2 and Z1 already hold their final output value for the loop and can be used as input parameters for a multiplication of the following loop.

This means that the first multiplication in every loop iteration performed after `montk1` has been executed, is a multiplication with the values of the registers X2 and Z1 as multiplicands. The second multiplication uses then X1 and Z2 as multiplicands. For this reason, the programs are executed in their variant forms every time the previous program execution has been done for `montk1`. As mentioned above, when the programs are executed in their normal form, the first multiplication is performed using X1 and Z2 as multiplicands and thus, the second uses X2 and Z1 as multiplicands.

The variations of both programs take place during the clock cycles 7, 8, 9, 18, 22, and 25. With these variations, it is possible to execute every loop iteration in a total of 54 clock cycles only.

Flow chart diagrams describing both re-designed programs (Figure 5.3 for `montk1` and Figure 5.4 for `montk0`) and their variations (Figure 5.5 for `montk1` and Figure 5.6 for `montk0`) are outlined in the same way as those presented in Chapter 4. For this reason, their description is very similar to the one presented in 4.3.3. Nevertheless small details result in important differences and it is thus worth providing a complete description:

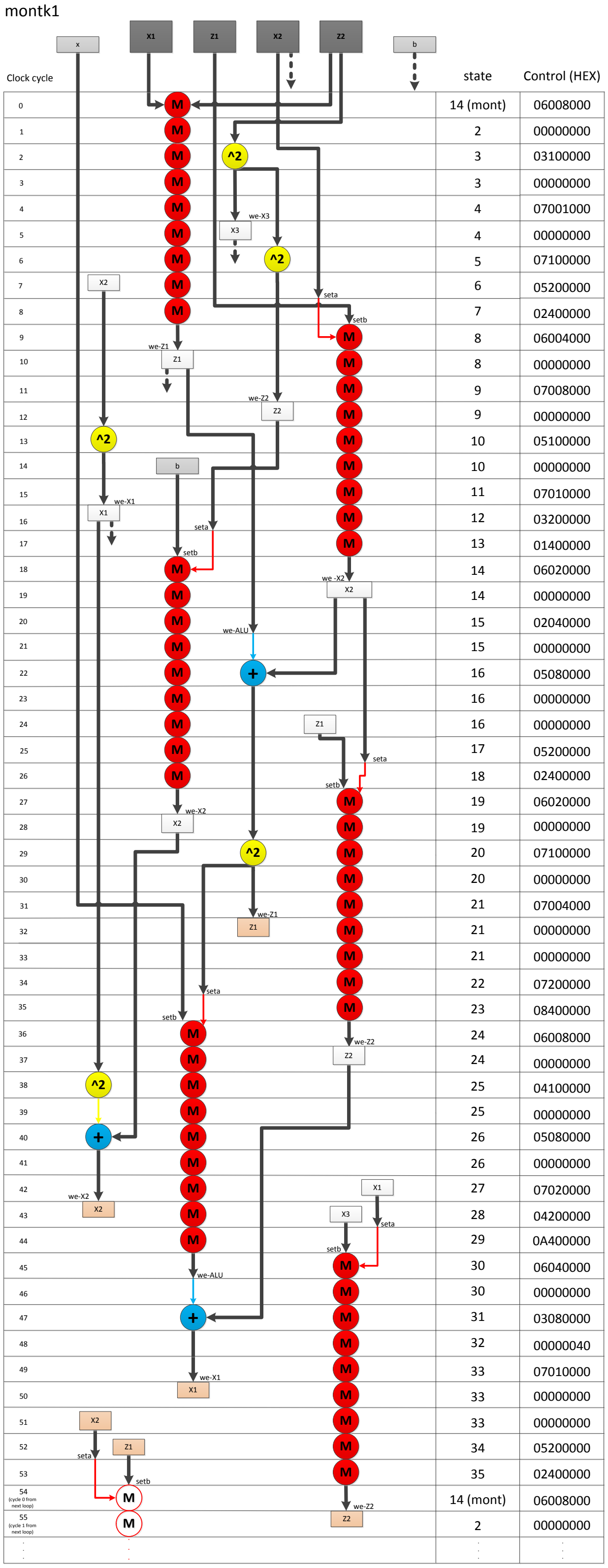
- On the left side of the diagrams the clock cycles are listed. Each loop consists now of 54 cycles only.
- The column named 'state' lists the states of the program during the loop. The state described in the cycle 0 (state = 14 (mont) in every chart) is part of the program `mont`. `mont` is the initialization program and it is always entered in its state = 14 for the first cycle of each loop. During this clock cycle, the first multiplication process for a loop starts and the Controller decides, if a loop for a bit '1' or for a bit '0' should be entered: if the Controller's input signal `is_set` has the value '1', state = 2 of `montk1` is entered in cycle 1 of the loop. If `is_set` has the value '0', state = 2 of `montk0` is entered in the cycle 1 of the loop.
- On the right side of the flow chart the values of the Controller's signal `cntr` are listed. They are given as hexadecimal numbers, as described in the VHDL file.
- The actions denoted as `we-` represent a *word enable* operation performed by the indicated register.
- The red units with the label 'M' represent the activity of the Multiplier. All units are displayed with a solid red color and this represents the current activity of the Multiplier. In contrast to the diagrams displayed in the previous chapter, there are no units displayed with a transparent red color. This means that there are no "non-operating" states of the Multiplier in these loops; this entity is constantly performing operations. Each loop contains 6 multiplications.

  The non filled units with the label 'M', placed in the cycles 54 and 55 of each loop represent multiplications which belong to the following loop.
- The yellow units with the label '^2' represent the squaring operation of an element of $GF(2^{233})$ performed by the ALU. Each loop contains 5 squaring operations. Each squaring operation needs one clock cycle.
- The blue units with the label '+' represent the addition of two elements of $GF(2^{233})$ performed by the ALU. This operation needs two clock cycles for the calculation of the sum: in the first cycle the first operand is saved in the inner register of the ALU

(when its *word enable* signal is set to ‘1’). In the second clock cycle the second operand is directly added to the first if its *xor enable* signal is set to ‘1’. There is a total of 3 additions performed in each loop.

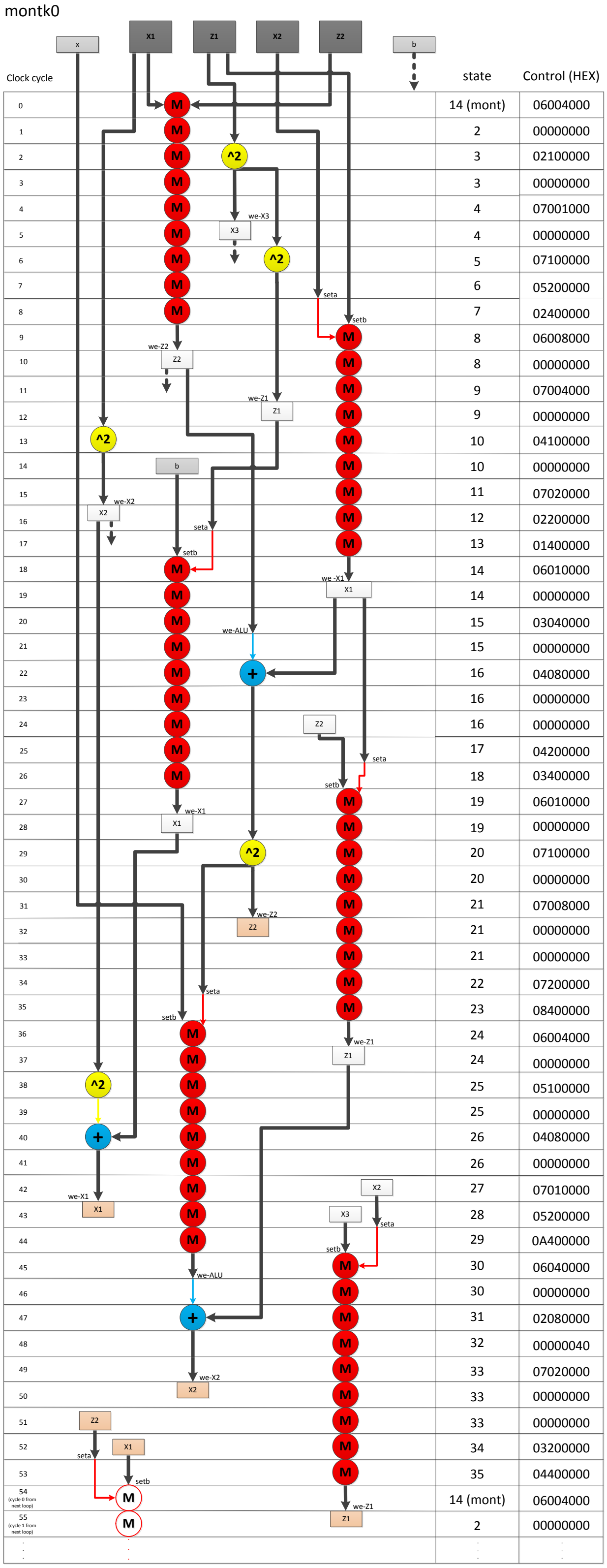
- All registers- X1, Z1, X2, Z2- are shown in grey colour when they are being used as input parameters at the beginning of the loop. They are white coloured when being used for saving intermediate values and beige coloured when they store the final values (output values) of the loop.

- The new registers X3 and X4 are displayed in white colour and are only used to store data inside the loop iteration. X4 is only used on the variant versions of both loops.



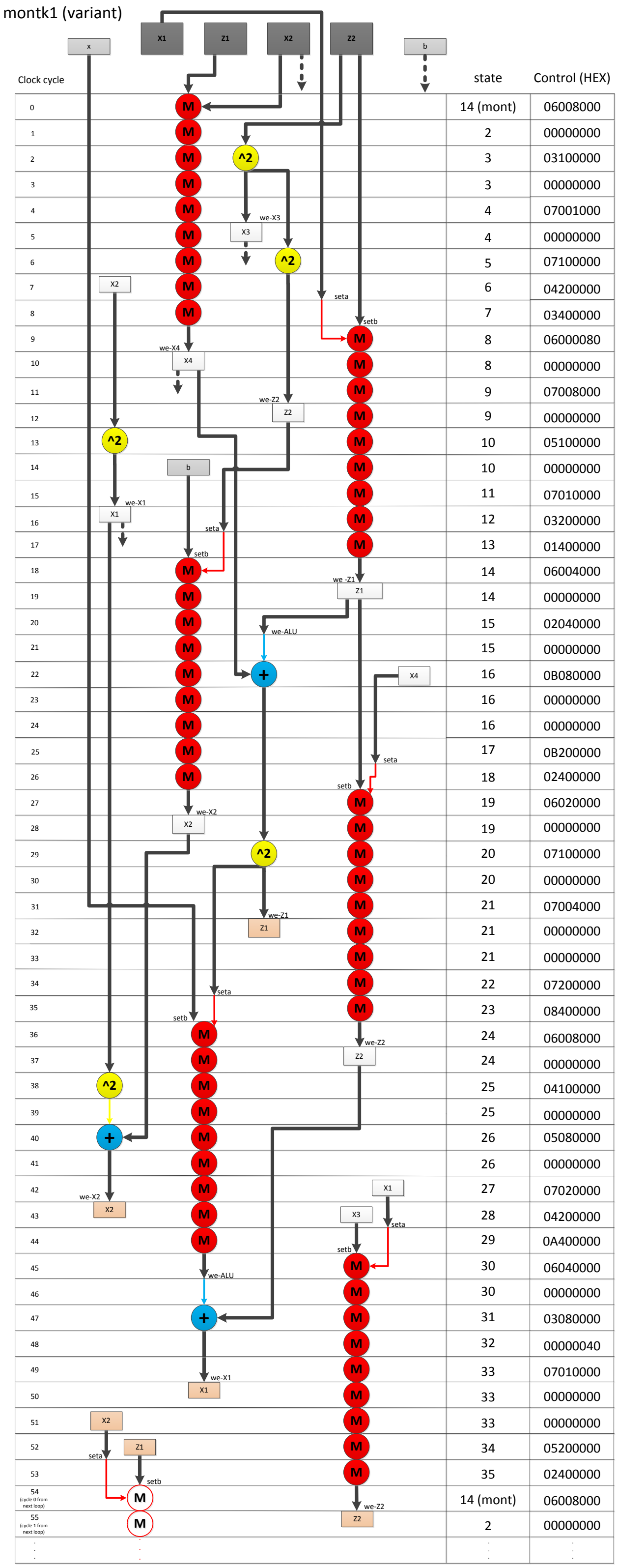
Clock cycle	state	Control (HEX)
0	14 (mont)	06008000
1	2	00000000
2	3	03100000
3	3	00000000
4	4	07001000
5	4	00000000
6	5	07100000
7	6	05200000
8	7	02400000
9	8	06004000
10	8	00000000
11	9	07008000
12	9	00000000
13	10	05100000
14	10	00000000
15	11	07010000
16	12	03200000
17	13	01400000
18	14	06020000
19	14	00000000
20	15	02040000
21	15	00000000
22	16	05080000
23	16	00000000
24	16	00000000
25	17	05200000
26	18	02400000
27	19	06020000
28	19	00000000
29	20	07100000
30	20	00000000
31	21	07004000
32	21	00000000
33	21	00000000
34	22	07200000
35	23	08400000
36	24	06008000
37	24	00000000
38	25	04100000
39	25	00000000
40	26	05080000
41	26	00000000
42	27	07020000
43	28	04200000
44	29	0A400000
45	30	06040000
46	30	00000000
47	31	03080000
48	32	00000040
49	33	07010000
50	33	00000000
51	33	00000000
52	34	05200000
53	35	02400000
54 (cycle 0 from next loop)	14 (mont)	06008000
55 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

Figure 5.3: Flow chart diagram of `montk1` re-designed.



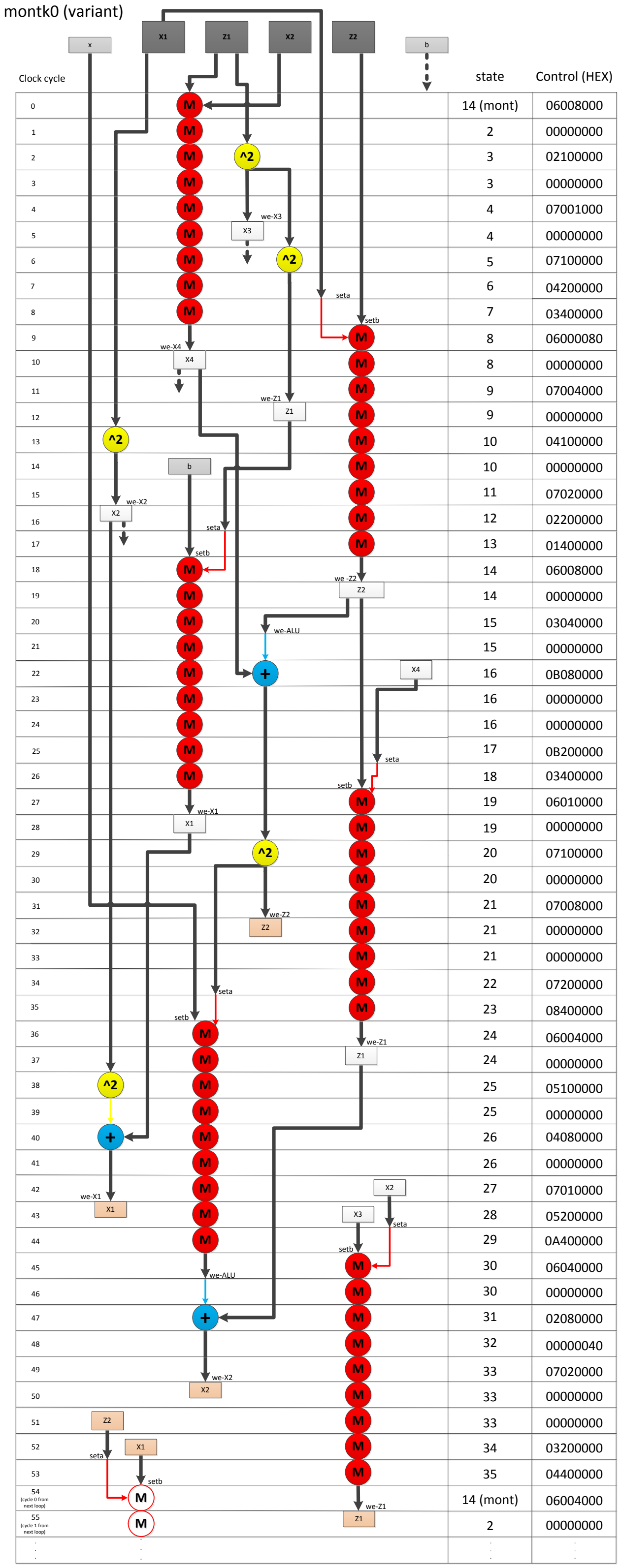
Clock cycle	state	Control (HEX)
0	14 (mont)	06004000
1	2	00000000
2	3	02100000
3	3	00000000
4	4	07001000
5	4	00000000
6	5	07100000
7	6	05200000
8	7	02400000
9	8	06008000
10	8	00000000
11	9	07004000
12	9	00000000
13	10	04100000
14	10	00000000
15	11	07020000
16	12	02200000
17	13	01400000
18	14	06010000
19	14	00000000
20	15	03040000
21	15	00000000
22	16	04080000
23	16	00000000
24	16	00000000
25	17	04200000
26	18	03400000
27	19	06010000
28	19	00000000
29	20	07100000
30	20	00000000
31	21	07008000
32	21	00000000
33	21	00000000
34	22	07200000
35	23	08400000
36	24	06004000
37	24	00000000
38	25	05100000
39	25	00000000
40	26	04080000
41	26	00000000
42	27	07010000
43	28	05200000
44	29	0A400000
45	30	06040000
46	30	00000000
47	31	02080000
48	32	00000040
49	33	07020000
50	33	00000000
51	33	00000000
52	34	03200000
53	35	04400000
54 (cycle 0 from next loop)	14 (mont)	06004000
55 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

Figure 5.4: Flow chart diagram of `montk0` re-designed.



Clock cycle	state	Control (HEX)
0	14 (mont)	06008000
1	2	00000000
2	3	03100000
3	3	00000000
4	4	07001000
5	4	00000000
6	5	07100000
7	6	04200000
8	7	03400000
9	8	06000080
10	8	00000000
11	9	07008000
12	9	00000000
13	10	05100000
14	10	00000000
15	11	07010000
16	12	03200000
17	13	01400000
18	14	06004000
19	14	00000000
20	15	02040000
21	15	00000000
22	16	0B080000
23	16	00000000
24	16	00000000
25	17	0B200000
26	18	02400000
27	19	06020000
28	19	00000000
29	20	07100000
30	20	00000000
31	21	07004000
32	21	00000000
33	21	00000000
34	22	07200000
35	23	08400000
36	24	06008000
37	24	00000000
38	25	04100000
39	25	00000000
40	26	05080000
41	26	00000000
42	27	07020000
43	28	04200000
44	29	0A400000
45	30	06040000
46	30	00000000
47	31	03080000
48	32	00000040
49	33	07010000
50	33	00000000
51	33	00000000
52	34	05200000
53	35	02400000
54 (cycle 0 from next loop)	14 (mont)	06008000
55 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

Figure 5.5: Flow chart diagram of `montk1` in its variant form.

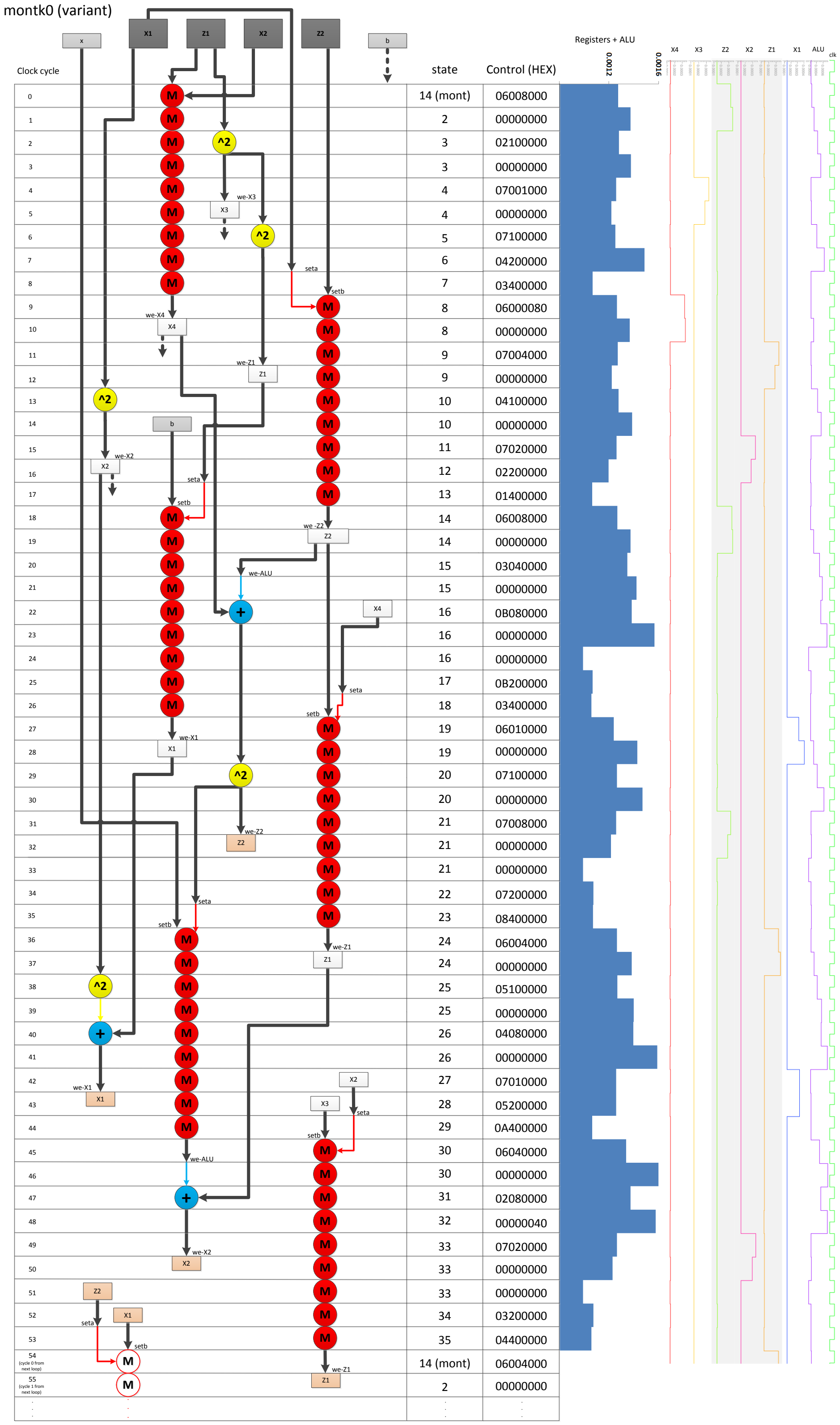


Clock cycle	state	Control (HEX)
0	14 (mont)	06008000
1	2	00000000
2	3	02100000
3	3	00000000
4	4	07001000
5	4	00000000
6	5	07100000
7	6	04200000
8	7	03400000
9	8	06000080
10	8	00000000
11	9	07004000
12	9	00000000
13	10	04100000
14	10	00000000
15	11	07020000
16	12	02200000
17	13	01400000
18	14	06008000
19	14	00000000
20	15	03040000
21	15	00000000
22	16	0B080000
23	16	00000000
24	16	00000000
25	17	0B200000
26	18	03400000
27	19	06010000
28	19	00000000
29	20	07100000
30	20	00000000
31	21	07008000
32	21	00000000
33	21	00000000
34	22	07200000
35	23	08400000
36	24	06004000
37	24	00000000
38	25	05100000
39	25	00000000
40	26	04080000
41	26	00000000
42	27	07010000
43	28	05200000
44	29	0A400000
45	30	06040000
46	30	00000000
47	31	02080000
48	32	00000040
49	33	07020000
50	33	00000000
51	33	00000000
52	34	03200000
53	35	04400000
54 (cycle 0 from next loop)	14 (mont)	06004000
55 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

Figure 5.6: Flow chart diagram of montk0 in its variant form.

The operations performed by the ALU and the `we-` operations performed by the registers demand a similar amount of power when they are executed. The ALU consumes about 0.6 mW over two clock cycles when it performs a squaring or an addition. When a `we-` operation is performed, the ALU and registers consume about 0.4 mW over two clock cycles. The sequence of all arithmetic and `we-` operations executed by each program has been ordered in such way, that there is always at least one clock cycle between the execution of two of these operations. With the operation sequence implemented this way, the amount of power consumed by the ALU and registers together has constantly a value between 1.3 and 1.6 mW.

As an example, Figure 5.7 shows the flow chart diagram for `montk0` when executed in its variant form. On the right side of the diagram, simulated PTs of the ALU and each register during the execution of `montk0` in its variant version are displayed. The sum of the power consumed by all registers and the ALU together is also displayed as one PT. It can be seen, that the power consumed by these seven FUs during the program execution has constantly a value between 1.3 and 1.6 mW when any of these FUs is active. The complete ECC design consumes a regular amount of power during the loop execution.



Clock cycle	state	Control (HEX)
0	14 (mont)	06008000
1	2	00000000
2	3	02100000
3	3	00000000
4	4	07001000
5	4	00000000
6	5	07100000
7	6	04200000
8	7	03400000
9	8	06000080
10	8	00000000
11	9	07004000
12	9	00000000
13	10	04100000
14	10	00000000
15	11	07020000
16	12	02200000
17	13	01400000
18	14	06008000
19	14	00000000
20	15	03040000
21	15	00000000
22	16	0B080000
23	16	00000000
24	16	00000000
25	17	0B200000
26	18	03400000
27	19	06010000
28	19	00000000
29	20	07100000
30	20	00000000
31	21	07008000
32	21	00000000
33	21	00000000
34	22	07200000
35	23	08400000
36	24	06004000
37	24	00000000
38	25	05100000
39	25	00000000
40	26	04080000
41	26	00000000
42	27	07010000
43	28	05200000
44	29	0A400000
45	30	06040000
46	30	00000000
47	31	02080000
48	32	00000040
49	33	07020000
50	33	00000000
51	33	00000000
52	34	03200000
53	35	04400000
54 (cycle 0 from next loop)	14 (mont)	06004000
55 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

Figure 5.7: Flow chart diagram of `montk0` with corresponding PTs of some FUs of the re-designed ECC implementation.

Special Execution of the First Loop Iteration

Besides the re-designs made for the programs `montk1` and `montk0`, programs `montk1pre` and `montk0pre` are additionally implemented in the block Controller. During the execution of the kP-operation, one of these two programs is executed when the main loop iteration is performed for the first time. According to the Montgomery kP-algorithm, the first main loop iteration takes place for processing the key bit k_{l-2}, whereby l is the length of k in bits.

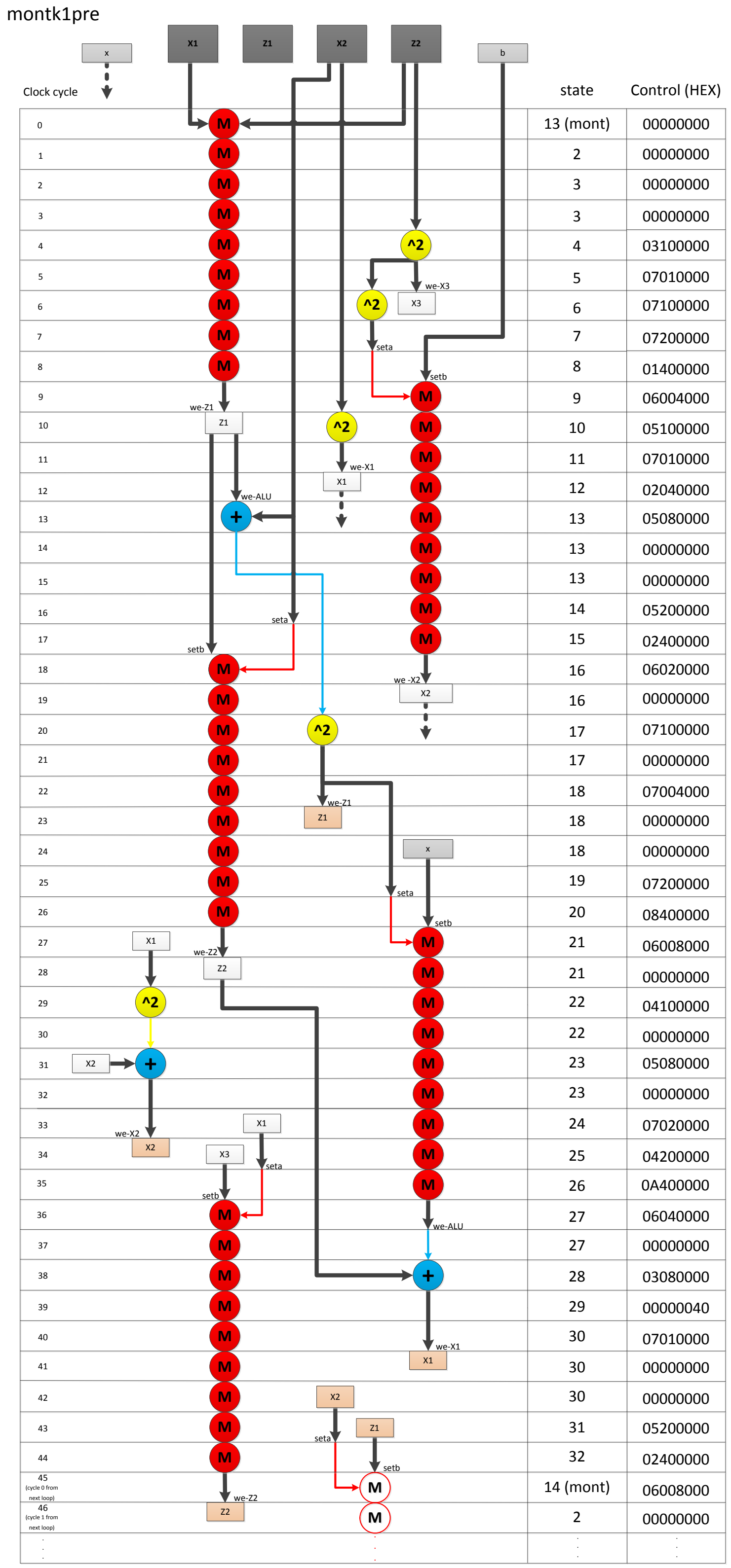
Depending on the value of the key bit k_{l-2}, `montk1pre` or `montk0pre` are executed after the initialization phase. These programs perform the loop iterations with several changes in comparison to the main programs `montk1` and `montk0`. These changes were made under the observation that the initialized value of register Z1 is 1. Given this fact, some of the operations in the first main loop iteration of the algorithm have results that can be known before their computation. For example, any multiplication that uses register Z1 with value 1 as a multiplicand results in the value of the other multiplicand. In a similar way, the squaring of register Z1 results in 1 as well: $\text{Z1}^2 = 1^2 = \text{Z1}$.
This means that some operation executions can be skipped. This reduces the time and energy consumption of this loop execution. It also hinders the performance of FSA attacks using the first main loop execution as a template, since this execution differs significantly from the rest.

In this thesis, the first iteration of the algorithm's main loop was simplified. Its execution time needs 9 clock cycles less than the rest of the loops. In the rest of this section, a detailed description of the implemented programs for the execution of the first loop of the kP-operation is given.

The first loop iteration can be simplified differently depending on the value of the key bit being processed. When $k_{l-2} = 1$, Z1=1 is used once as a multiplicand. When $k_{l-2} = 0$, Z1=1 is used once as a multiplicand and once as input parameter for a squaring operation. The result of this squaring operation is as well 1 and it is then squared again. The result of the second squaring, which is as well 1, is used as a multiplicand to perform another multiplication. This means that if the first loop iteration is performed for processing the key bit $k_{l-2} = 1$, the iteration can be performed with one multiplication less than usual. If it is performed for processing $k_{l-2} = 0$, the iteration can be performed with two multiplications and two squaring operations less than usual. If all these operation executions would be skipped when $k_{l-2} = 0$, the loop execution would be shorter (unbalanced), than when $k_{l-2} = 1$. Under this observation, not all possible optimizations were made for the program `montk1pre`.

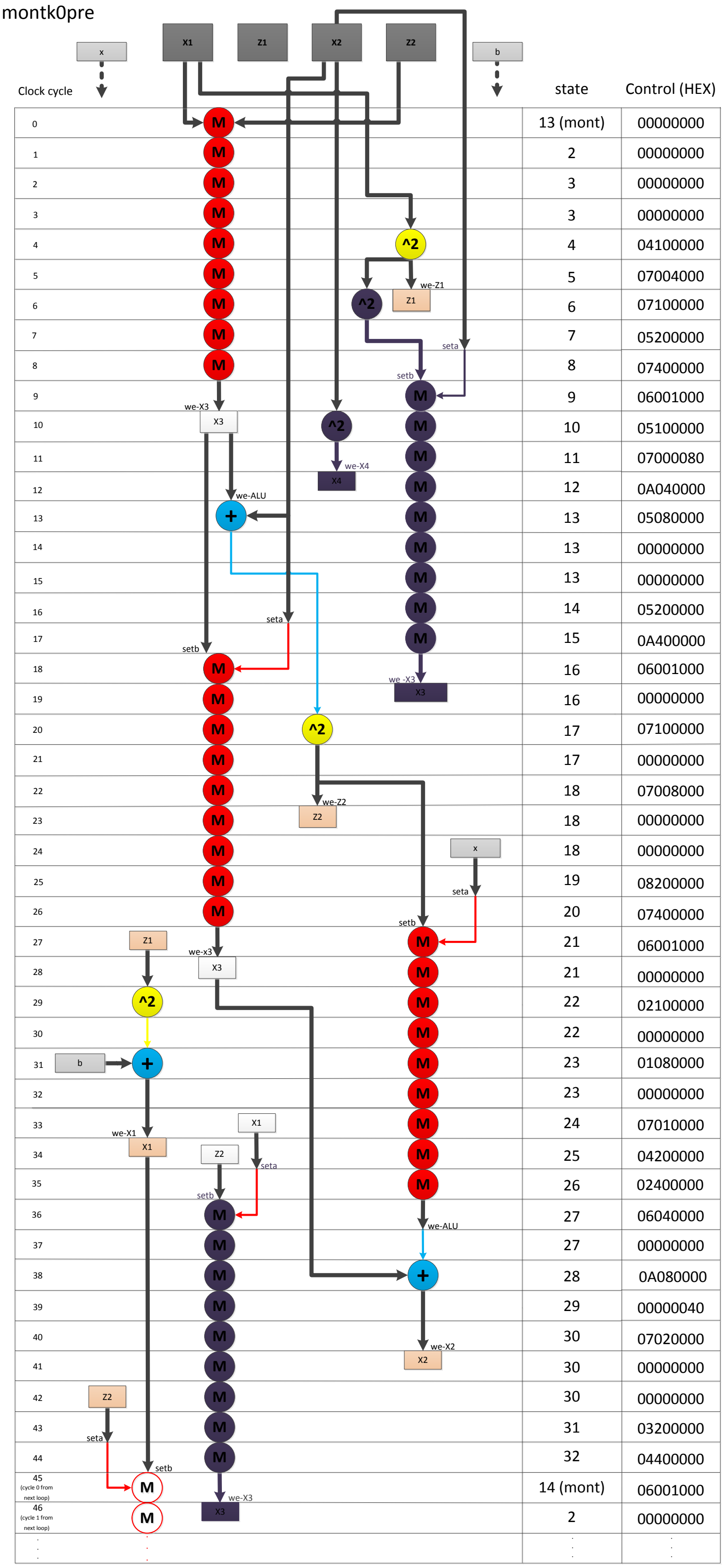
The flow chart diagrams in Figures 5.8 and 5.9 describe the operation flow for both programs. The programs `montk1pre` and `montk0pre` have the following differences in comparison to `montk0` and `montk1`:

- Each program performs only 5 multiplications, i.e. one multiplication less than `montk1` and `montk0`. This loop iteration consists of only 45 clock cycles.
- Register Z1 is not used as a direct input parameter of the loop. Its value is first overwritten in cycle 5 for `montk0pre` and in cycle 9 for `montk1pre`.
- The state described in cycle 0 of each diagram (state = 13 (mont) for both cases) is part of the initialization program `mont`. This state is only entered once after the initialization phase. In this state, it is known that the first iteration of the main loop, i.e. for the key bit k_{l-2}, should be processed as next. The first multiplication process for this loop starts with the input values of the registers X1 and Z2 and the Controller decides if the program `montk0pre` or `montk1pre` should be entered.
- Program `montk0pre` executes dummy operations, which are displayed in a dark blue colour in its corresponding flow chart diagram (see Figure 5.9). The results of these operations are not used. These operations are only executed in order to achieve balance between the time and power consumption of both programs `montk1pre` and `montk0pre`. This makes the execution of both programs indistinguishable, independently form the key bit value being processed.
- During the last two clock cycles, 43 and 44, the input values for the first multiplication of the following loop are set. Here `montk0pre` and `montk1pre` operate the same way as described for `montk0` and `montk1`.
- Clock cycle 45 on the diagrams displays the first clock cycle of the next loop. The state described in this cycle (state = 14 (mont)) is part of the program `mont`. In this state it is decided if state = 2 of `montk1` or of `montk0` is entered.



Clock cycle	state	Control (HEX)
0	13 (mont)	00000000
1	2	00000000
2	3	00000000
3	3	00000000
4	4	03100000
5	5	07010000
6	6	07100000
7	7	07200000
8	8	01400000
9	9	06004000
10	10	05100000
11	11	07010000
12	12	02040000
13	13	05080000
14	13	00000000
15	13	00000000
16	14	05200000
17	15	02400000
18	16	06020000
19	16	00000000
20	17	07100000
21	17	00000000
22	18	07004000
23	18	00000000
24	18	00000000
25	19	07200000
26	20	08400000
27	21	06008000
28	21	00000000
29	22	04100000
30	22	00000000
31	23	05080000
32	23	00000000
33	24	07020000
34	25	04200000
35	26	0A400000
36	27	06040000
37	27	00000000
38	28	03080000
39	29	00000040
40	30	07010000
41	30	00000000
42	30	00000000
43	31	05200000
44	32	02400000
45 (cycle 0 from next loop)	14 (mont)	06008000
46 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

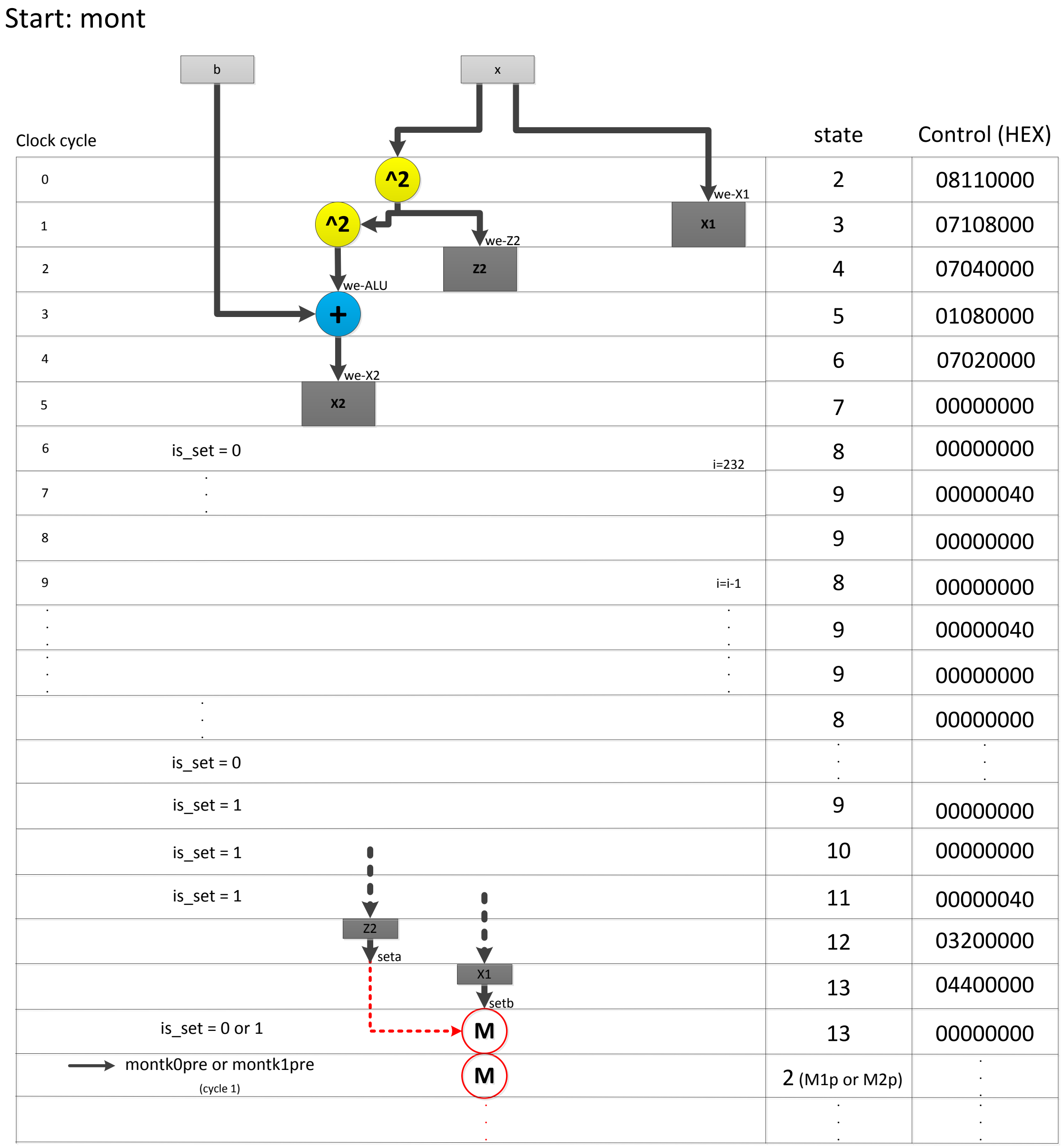
Figure 5.8: Flow chart diagram of `montk1pre`.



Clock cycle	state	Control (HEX)
0	13 (mont)	00000000
1	2	00000000
2	3	00000000
3	3	00000000
4	4	04100000
5	5	07004000
6	6	07100000
7	7	05200000
8	8	07400000
9	9	06001000
10	10	05100000
11	11	07000080
12	12	0A040000
13	13	05080000
14	13	00000000
15	13	00000000
16	14	05200000
17	15	0A400000
18	16	06001000
19	16	00000000
20	17	07100000
21	17	00000000
22	18	07008000
23	18	00000000
24	18	00000000
25	19	08200000
26	20	07400000
27	21	06001000
28	21	00000000
29	22	02100000
30	22	00000000
31	23	01080000
32	23	00000000
33	24	07010000
34	25	04200000
35	26	02400000
36	27	06040000
37	27	00000000
38	28	0A080000
39	29	00000040
40	30	07020000
41	30	00000000
42	30	00000000
43	31	03200000
44	32	04400000
45 (cycle 0 from next loop)	14 (mont)	06001000
46 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

Figure 5.9: Flow chart diagram of `montk0pre`.

The initialization process of the kP-operation also went through modifications and was simplified do to the fact that no initialized value needs to be assigned to register Z1 any more. The flow chart diagram in Figure 5.10 describes the re-designed initialization process. Register X1 is initialized in cycle 0 with the value of register x; register Z2 is initialized in cycle 1 with the value x^2; register X2 is initialized in cycle 4 with the value x^4+ b.



Clock cycle		state	Control (HEX)
0		2	08110000
1		3	07108000
2		4	07040000
3		5	01080000
4		6	07020000
5		7	00000000
6	is_set = 0	8	00000000
7		9	00000040
8		9	00000000
9		8	00000000
.		9	00000040
.		9	00000000
		8	00000000
	is_set = 0	.	.
	is_set = 1	9	00000000
	is_set = 1	10	00000000
	is_set = 1	11	00000040
		12	03200000
		13	04400000
	is_set = 0 or 1	13	00000000
		2 (M1p or M2p)	.
		.	.

Figure 5.10: Flow chart of the re-designed initialization part of the Montgomery kP-algorithm.

5.2 Observations

This section lists observations made on the re-designed IHP ECC implementation using simulated PTs of the complete design and its FUs separately. These observations show improvements regarding the resistance of the re-designed implementation against SCA.

1) Boundaries between the loops are not easily identifiable

Figure 5.11 shows a part of a PT of the re-designed ECC implementation during four main loop executions.

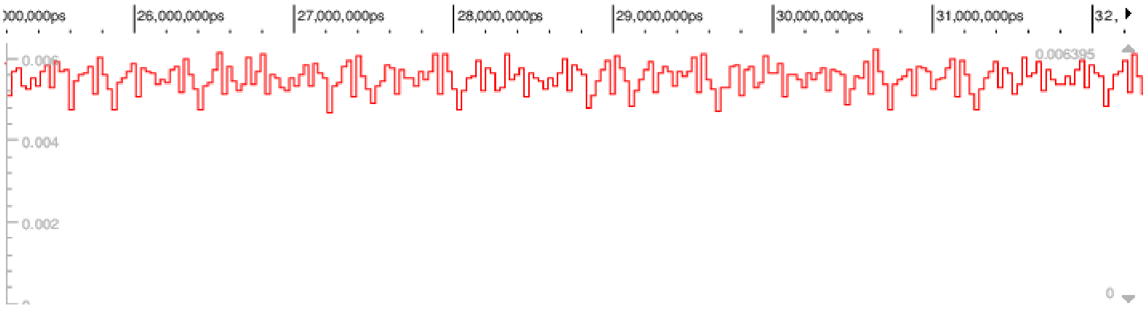


Figure 5.11: Part of a PT simulated for the re-designed ECC implementation.
The graph shows power measurements (in W) over a period of time (in ps). Compared to the PT of the original ECC implementation (see Figure 4.13), there are no dips or plateaus along the curve.

The simulated PT of the re-designed ECC implementation contains no dips or plateaus along its curve while performing the main loop iterations. This way, it is not easy to identify boundaries between the single loops, i.e. between the slots of the PT.

In the re-designed implementation, the Multiplier is constantly active during the loop iterations. The sequence of the squaring, addition and `we-` operations performed in these loops has been implemented in such way, that each operation is performed at least every two clock cycles. These three operations consume a similar amount of power over two clock cycles each time they are executed. By always leaving at least one clock cycle between the execution of each operation, the power consumption of the complete design remains regular along the loop iteration.

2) Balanced implementation of the ECC design

Each loop iteration is dependent on the value of the processed key bit. Nevertheless, the same operations are performed at the same time for every iteration (see details in p. 76).

3) The first loop iteration gives no information about the processed key bit

Figure 5.12 shows a part of a simulated PT. It corresponds to the execution of the first loop and its following loops. The programs montk1pre and montk0pre execute the first loop iteration. It was mentioned in 5.1.3, that both programs have been designed in the same way. They perform the same operations in the same sequence. So both programs produce a similar PT for the first loop.

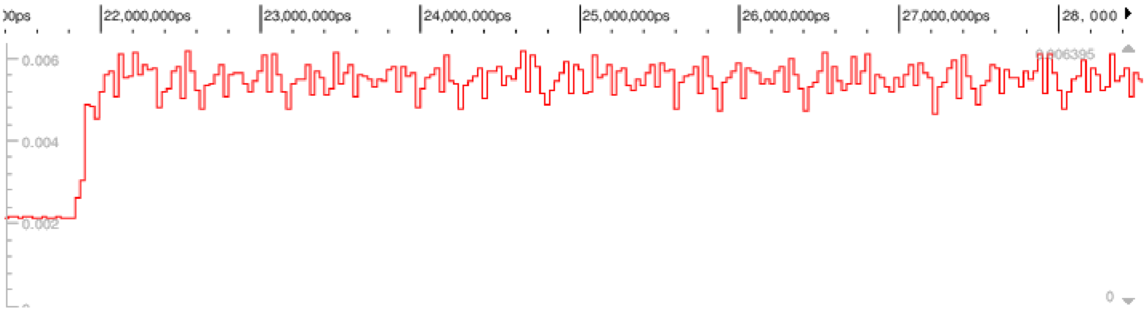


Figure 5.12: PT simulated at the beginning of the execution of the kP-operation.
The graph shows power measurements (in W) over a period of time (in ps). No low power consumption can be detected at the beginning of the curve.

With the modifications made for the first loop iteration, the value of k_{l-2} cannot be easily extracted as it was feasible for the original ECC implementation. A comparison can be made with Figure 4.18, which shows the PTs simulated during the first loop iteration performed by the original implementation. By executing the first loop iteration in this way, the slot corresponding to the processing of k_{l-2}, i.e. the initial data execution, cannot be used as a template for attacks based on FI, for example the FSA attack described in section 3.2.2.

Besides protecting the identity of the key bit k_{l-2} and providing protection against the FSA attack, the PT shows a small plateau near its beginning. This plateau is caused due to the fact that during clock cycles 4, 5 and 6 of these programs, one squaring, one we- and another squaring operation is performed. For the rest of the programs, there is at least one clock cycle between each operation. This bigger amount of operations

performed in such a small amount of time causes a higher power consumption of the entire design during this period.
This characteristic also makes the analysis of the complete PT more difficult since it is not clearly understandable, at which moment the first key bit has stopped being processed and what this bigger power consumption means.

4) Chip area and power consumption of the re-designed implementation

The modifications made on the design had an obvious effect on its chip area. Two new registers (X3 and X4) are used and the block Controller has been extended through the programs `montk1pre` and `montk0pre`. The programs for `montk1` and `montk0` have also been extended since conditional instructions were added to them. On the other hand, the initialization phase of the algorithm was simplified. With these modifications, a first re-design of the implemented Montgomery kP-algorithm was done.

After the first re-design, a few functionalities of the blocks Multiplier and ALU were identified, which were no longer used. The Multiplier had additional input signals to control its functionality while it was in a non-operating state (illustrated by the units with transparent red color in Figures 4.11 and 4.12). The ALU had one input signal and a corresponding internal state, with which its internal register was set to the value 1. This was only used in the initialization phase of the Montgomery kP-algorithm (see clock cycles 5 and 6 in Figure 4.17). These and other unused functionalities[1] were removed from the design. Table 5.3 shows the chip area of the original IHP hardware accelerator for ECC and of the first and final re-designed implementation.

Table 5.3: Total cell area of the different versions of the ECC implementation

ECC version	**original**	**first re-design**	**final re-design**
area	0.260258 mm^2	0.284256 mm^2	0.274844 mm^2

The final re-designed implementation has an area 5.6% bigger than the original design. In a similar way, the complete power consumption of the implementation was affected by its modifications. In the re-designed version, a bigger number of operations is performed per loop and additional (conditional) instructions are executed during several clock cycles. On the other hand, the execution of each loop was reduced from 57 to only 54 clock cycles and the simplifications made in some FUs reduced the number of operations performed. Table 5.4 shows the power, time and energy consumption

[1] Unused lines of code were identified in the original ECC design.

of the first and final re-designed implementation in comparison to the original IHP ECC design. The execution time is calculated by the number of clock cycles taken for a complete execution of the kP-operation, with one clock cycle having a period of $T = 0.30$ ns.

Table 5.4: Power, time and energy consumption of the different versions of the ECC implementation.

ECC version	**original**	**first re-design**	**final re-design**
Power consumption	4.90 mW	5.51 mW	5.40 mW
Time of a kP-operation	0.41 ms	0.39 ms	0.39 ms
Energy consumption per kP-operation	2 μJ	2.13 μJ	2.09 μJ

5.3 Results of the Difference-of-Means Test

A horizontal DPA attack using the difference-of-means test was performed for the re-designed version of the IHP ECC implementation in the same way and using the same inputs as described in Chapter 4. This section describes the results of this attack.

The difference-of-means test was performed using simulated PTs for the following cases:

- case 1: for the kP-design processing the EC point $P1 = (x_1, y_1)$ as the input data using the scalar $k1$
- case 2: for the kP-design processing the EC point $P1 = (x_1, y_1)$ as the input data using the scalar $k2$

The values of these input parameters were the same as those used in Chapter 4 (see p. 38).

The test was performed for the following design blocks separately:

1. complete ECC design (ecc)
2. block Multiplier (mult)
3. block ALU
4. register x1 (X1)
5. register z1 (Z1)
6. register x2 (X2)
7. register z2 (Z2)
8. register x3 (X3)

9. register x4 (X4)

Additionally, the difference-of-means test was performed for the sum of the power consumptions of the registers X1, Z1, X2, Z2, X3, X4 and the block ALU.

For this test, only the PTs simulated during the processing of the scalar k were chosen. These corresponds to the slots of the PTs simulated during the execution of the Montgomery kP-algorithm's main loops. The slot corresponding to the first execution of the main loop was not included for this test[2]. Thus, a total of 230 slots were obtained from the simulated PT for performing this test.

The difference-of-means test was performed as follows:

1 - The investigated part of the PT was partitioned into 230 slots, 54 clock cycles each. Each slot was used as a separate curve that consists of 54 points. Figure 5.13 shows the first 8 regular slots of the simulated PT for case 1.

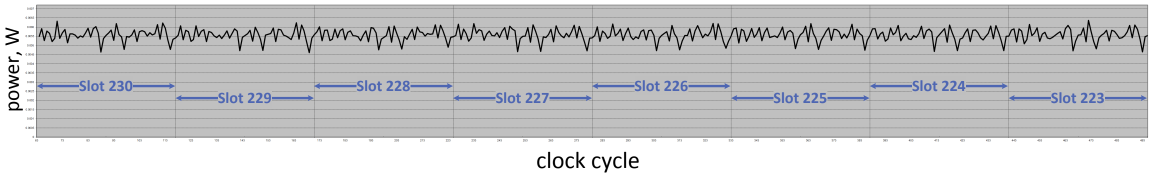


Figure 5.13: First 8 regular slots of the simulated PT for case 1.
The re-designed IHP kP-implementation processed the EC point $P1 = (x_1, y_1)$ as the input data using the scalar $k1$.

2 - The mean curve was calculated using 230 slots.

3 - The power value of the 1st point of the mean curve was compared with the power value of the 1st point of each slot. The first slot corresponded to the processing of the key bit $k_{l-3} = k_{230}$. The last slot corresponded to the processing of the key bit k_0. If the power value of the mean curve was higher than the value of the curve in the current slot, it was assumed that the slot corresponded to the key bit value '1', otherwise to '0'. Thus, the first key candidate was obtained.

4 - Step 3 was repeated for all other 53 points of the mean curve and the remaining 53 key candidates were obtained.

[2] The re-design ECC implementation performs the first loop iteration differently and in a shorter period of time as the rest of the loop iterations (see section 5.1.3)

The variance for all 54 points of the mean curve were calculated. Figure 5.14 shows all calculated variances for case 1 as a blue and all variances for case 2 as a red curve.

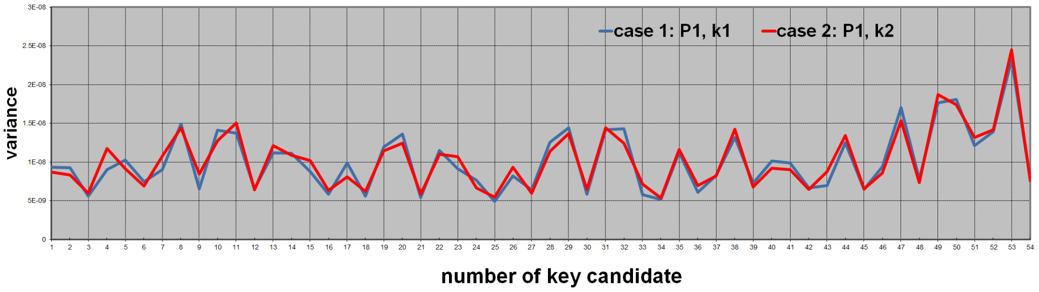


Figure 5.14: All calculated variances represented as a curve consisting of 54 points. The blue curve shows the variances calculated for case 1 and the red curve shows the variances calculated for case 2.

Both variance curves are similar, contrary to the variance curves obtained with the original ECC design (see Figures 4.3 and 4.4). Both curves in Figure 5.14 show a peak on point 53. Using the variance as criteria, this point can be identified as a potential SCA leakage source.

Each of the 54 obtained key candidates was compared with the key value that was actually processed. For each key candidate, the three different comparisons v1, v2 and v3 were made (see details in p. 42 of section 4.2). Figure 5.15 shows the relative correctnesses of the key extraction for case 1, whereby the white dotted curve corresponds to v1, the black dotted curve to v2 and the yellow dotted curve for v3. The red solid curve consists of the values of the best key candidates from all three models (v1, v2, v3) for each point.

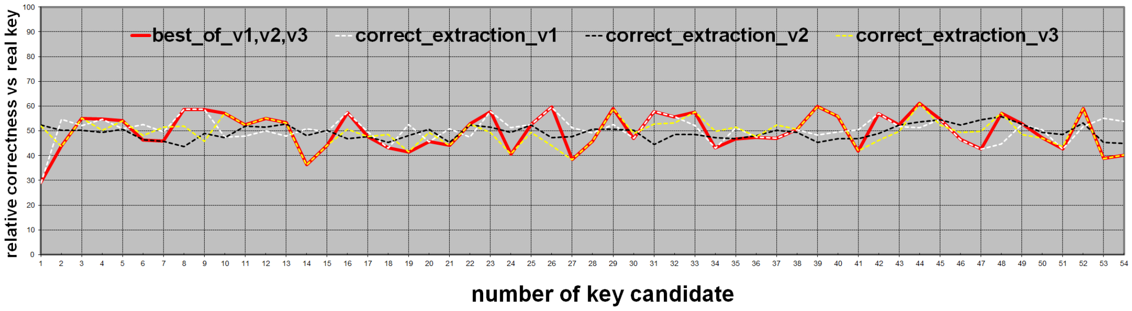


Figure 5.15: Relative correctness of the key extraction as a curve for case 1.

This process was done for case 2 as well. The results for both cases are shown in Figure 5.16. The red curve shows the results obtained for case 1 (the same red solid curve as shown in Figure 5.15) and the black curve shows the results obtained for case 2. The green curve corresponds to the ideal case.

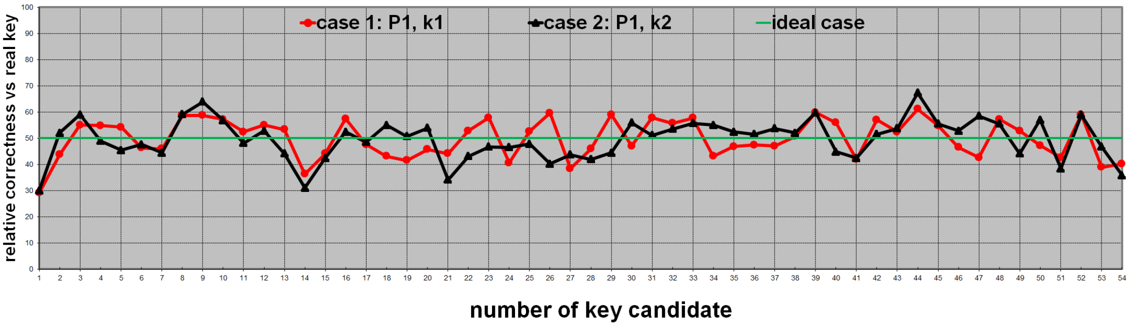


Figure 5.16: Relative correctness of the extraction of the key for each of the 54 key candidates as a curve. The red curve corresponds to case 1 and the black curve to case 2; the green curve shows the ideal case.

Both curves in Figure 5.16 have the same value at point 1. For both cases, the correctness of the key extraction for the key candidate 1 has a value of 70% (100-30=70%). No other key candidate reaches a value of 70% or higher for any of the two cases. In comparison to the original ECC design, the resistance against the performed DPA attack with the re-designed implementation is much stronger. The test performed on the original version gives five key candidates with a relative correctness higher than 85% (see p. 44).

The results of this difference-of-means test lead to the following conclusions about the re-designed IHP ECC implementation:

- The SCA leakage points identified in Chapter 4 have been successfully eliminated: the horizontal DPA attack using the difference-of-means test was not successful.
- The re-designed ECC implementation does not show strong SCA leakage sources.

Even though the difference-of-means test performed on the complete ECC design does not show strong SCA leakage sources, the test performed with the PTs of the individual blocks of the re-designed implementation shows many possible leakage sources. Figure 5.17 shows the relative correctness of the extraction of the key for each of the 54 key candidates for most blocks of the ECC design. Figure 5.17 consists of the following curves:

a) The relative correctness curve for the complete ECC design (ecc) is displayed in black colour. The red curve corresponds to the block Multiplier (mult).

b) The relative correctness curve for register X1 (x1) is displayed in blue. The black dotted curve displays the relative correctness of register Z1 (z1).

c) The relative correctness curve for register X2 (x2) is displayed in blue. The black dotted curve displays the relative correctness of register Z2 (z2).

d) The relative correctness curve for register X3 (x3) is displayed with the blue dotted curve. The orange curve displays the relative correctness of register X4 (x4).

e) The relative correctness curve for the ALU block (alu) is displayed in dark blue.

f) The relative correctness curve for the case that all six registers X1, Z1, X2, Z2, X3, X4 and the ALU are considered one block (all_register+alu). This means that the sum of the power consumption of these seven blocks was analysed in a difference-of-means test.

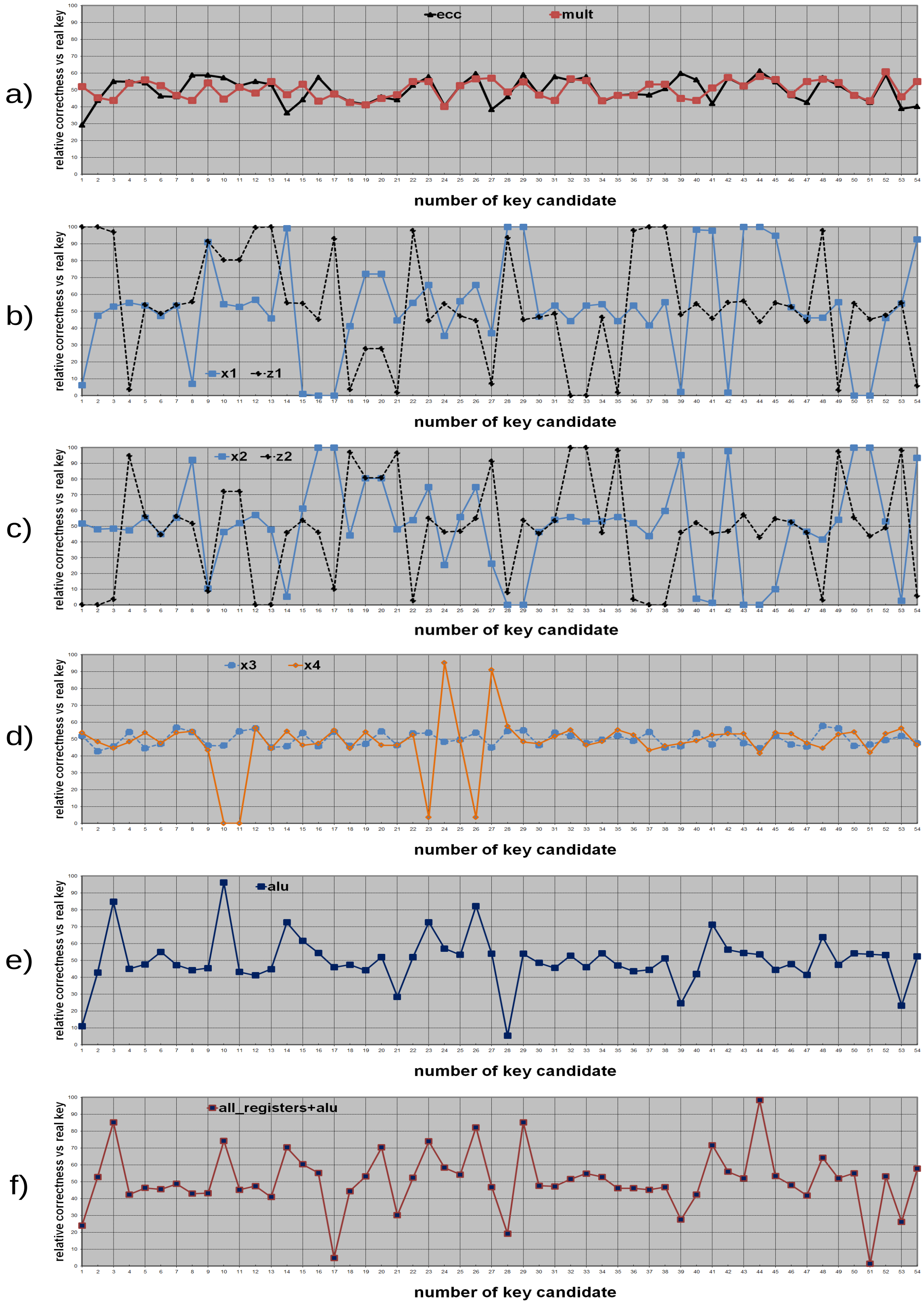


Figure 5.17: Relative correctness of the key extraction using PTs of the re-designed ECC implementations and its individual blocks.

The registers X1, X2, Z1, Z2, X4 and the ALU (Figure 5.17 b), c), d) e)) are the most insecure (leaking) blocks. The activity of the Multiplier (Figure 5.17 a)) cannot be assessed as a significant SCA leakage source any more.

An attacker does not have the possibility to measure the power consumption of individual blocks of a cryptographic implementation. For this reason, he cannot obtain the results displayed in Figure 5.17. These results are only known to the designers and should be considered for understanding how the power consumption of individual FUs in a cryptographic design could lead to SCA leakage sources.

5.4 Results of a Comparative Power Analysis Attack

To test the resistance of the re-designed ECC implementation against a different type of PA, a ComPA attack was made with power consumption measurements. The attack described in this section was successfully performed on the original ECC implementation [38]. The ECC implementation was run in an FPGA Spartan 6 from Xilinx and the FPGA's power consumption was measured. The setup and procedure for performing the power consumption measurements was the same as described in [38]. Measurements were made for the following cases:

- case 1: for the kP-design processing the EC point $P1 = (x_1, y_1)$ as the input data using the scalar $k1$
- case 2: for the kP-design processing the EC point $P1 = (x_1, y_1)$ as the input data using the scalar $k2$
- case 3: for the kP-design processing the EC point $P1 = (x_1, y_1)$ as the input data using the scalar $k3$

The following values were assigned to the EC point coordinates and scalars, here represented in hexadecimal notation:

x_1 = 181 856adc1e 7df13784 91fa736f 2d02e8ac f1b9425e b2b061ff 0e9e8246

y_1 = 89 fed47b79 6480499c baa86d8e b39457c4 9d5bf345 a0757e46 e2582de6

$k1$ = 93 919255fd 4359f4c2 b67dea45 6ef70a54 5a9c44d4 6f7f409f 96cb52cc

$k2$ = 93 919255fd 4359ffff ffffffff ffffffff ffffffff ffffffff ffffffff

$k3$ = ff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff

This means that the kP-operation was performed by the same ECC implementation using the same point $P1$ for all cases. This is a standard condition for performing a ComPA attack (see p. 26 in section 3.1.4). The keys $k1$, $k2$ and $k3$ have different values, whereby the first quarter of the key bits of $k1$ and $k2$ are the same, but the rest of $k2$ consists only of bits with the value '1'. $k3$ consists completely of key bits with value '1'. Figure 5.18 shows the PTs measured for all three cases. The uppermost trace corresponds to case 1, the middle trace to case 2 and the trace in the bottom corresponds to case 3.

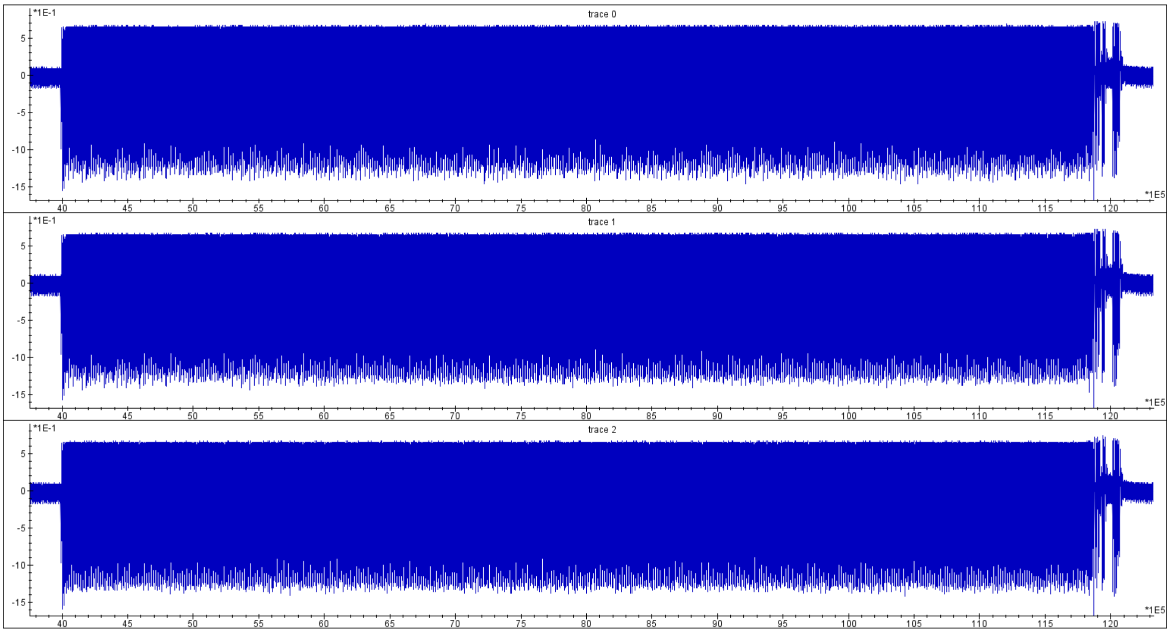


Figure 5.18: PTs measured from the re-designed ECC implementation executed on an FPGA. The Figure shows power measurements (in mW) over time (in ms). The uppermost trace corresponds to the execution of the kP-algorithm using $k1$, the trace in the middle was measured using $k2$ and the lowermost trace using $k3$.

The assumption made for ComPA is that if an implementation performs the same operations using the same input parameters, the difference between their measured PTs should consist only of the noise that was part of the measurements, i.e. the difference between both traces should be very small.

The first quarter of the key for case 1 and 2 were the same, this means that about the first 58 loop iterations for these two cases have been executed exactly in the same way, using the same input data. As soon as one key bit $k1_i$ differs from the corresponding key bit $k2_i$, the loop iterations are performed with different input and output parameters for all the following iterations. Consequently, the difference between the PTs measured for case 1 and case 2 should not give a small difference any more. The calculated difference between the PTs measured for case 1 and case 3 on the other hand should not have a small value since both keys used $k1$ and $k3$ are completely different. Figure 5.19 shows the calculated difference of the PTs measured for case 1 and 2 (see the middle curve) and the calculated difference of the PTs measured for case 1 and 3 (see the curve in the bottom). The curve at the top corresponds to the PT measured for case 1 and is shown in the picture as a length reference for the curves. It can be seen that the first quarter of the middle curve has a very small value. From the second quarter on, the difference is not small any more and has irregular values. The values along the lowest curve are never small and irregular all the time.

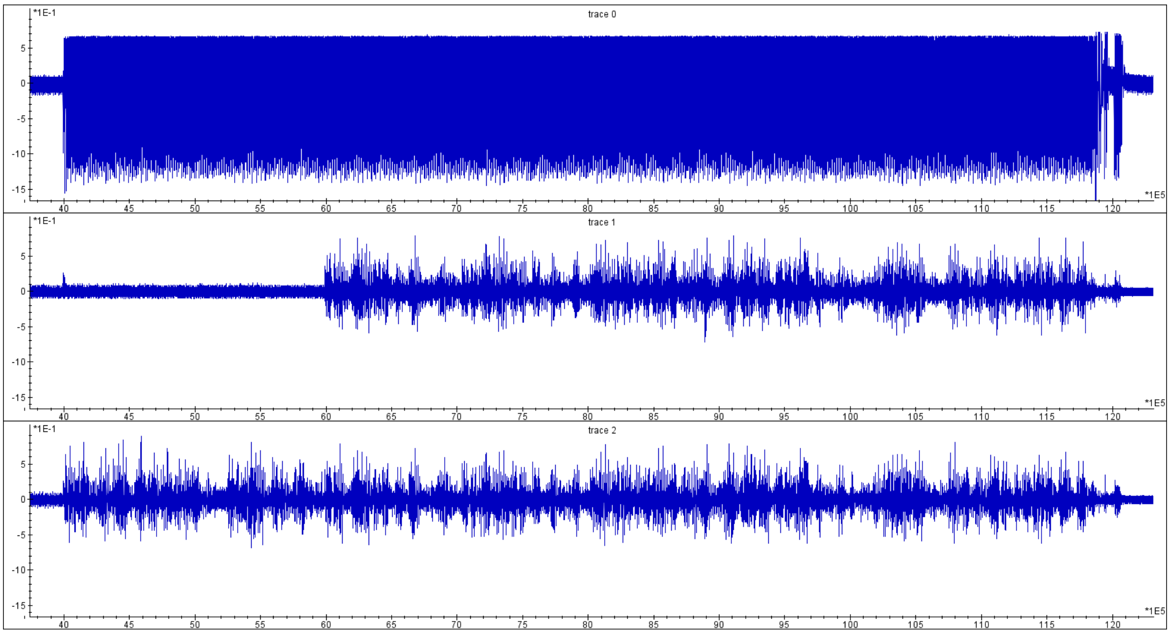


Figure 5.19: Calculated difference of the measured PTs.
The Figure shows power measurements (in mW) over time (in ms). The middle curve is the calculated difference for cases 1 and 2. The bottom curve shows the calculated difference for cases 1 and 3 (lowest curve). The curve at the top corresponds to the PT measured for case 1.

The results shown in Figure 5.19 prove that a ComPA attack can still be performed on the re-designed ECC implementation. Further DPA countermeasures, based for example on randomization, should be considered to achieve a more secure ECC implementation.

6 Verification of the Re-design Methodology

In order to verify if the re-design ideas described in Chapter 5 could also make other ECC implementations more resistant against SCA, the same methodology was used to re-design the IHP hardware accelerator for ECC with a 6-clock-cycle multiplier. This chapter describes shortly the re-design process done for this ECC implementation. The results of a horizontal DPA attack performed using the difference-of-means test confirm that the re-design methodology can be used for improving the resistance of further ECC implementations against SCA.

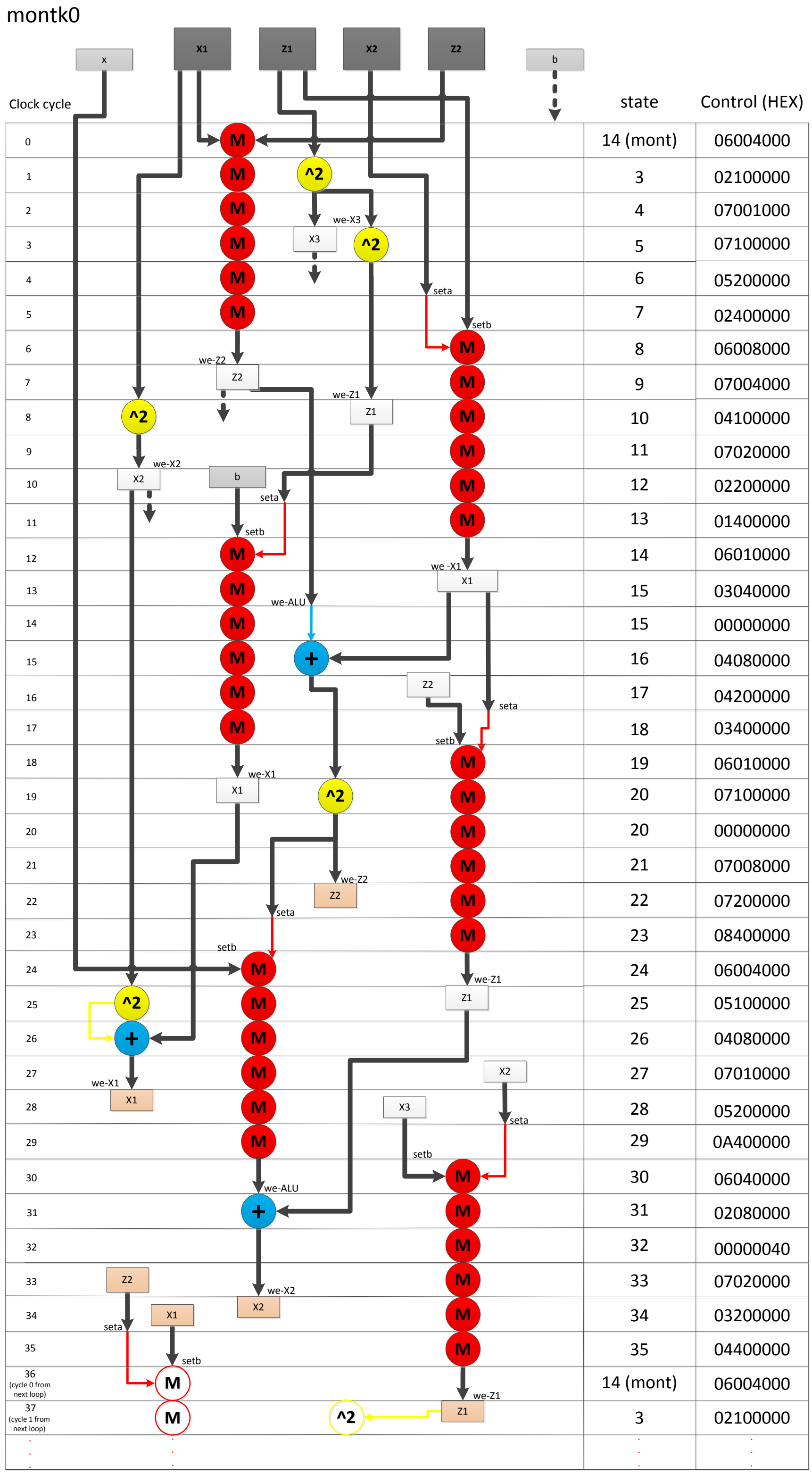
6.1 Re-design of the ECC Implementation with the 6-clock-cycle Multiplier

The IHP hardware accelerator for ECC with the 6-clock-cycle multiplier was proven to be vulnerable against a horizontal DPA attack using the difference-of-means test [35]. This design consisted of a similar architecture as the one described in Chapter 4, but with one significant difference: the $GF(2^{233})$ element's multiplication was implemented using the *iterative Winograd* multiplication method, which needs 6 clock cycles for the calculation of one product. For the processing of one key bit, this design needed 41 clock cycles. Both implementations have the same bus architecture and all other ECC blocks operate in the same way (see section 4.3).

6.1.1 Re-design of the Main Loop

Since the ECC design with the 6-clock-cycle multiplier has the same architecture as the one described in section 4.3.1, the re-design methodology presented in Chapter 5 could be implemented in the same way as it was done for the ECC design with the 9-clock-cycle multiplier. The only difference to consider when doing this was the fact that the processing of each bit, i.e. each loop iteration of the Montgomery kP-algorithm, could be performed in a shorter period of time. As described above, the Multiplier integrated in this ECC design needs 6 clock cycles for performing one multiplication. The Montgomery kP-algorithm's main loop includes 6 multiplications. This means that at least 36 clock cycles are needed in order to perform one complete loop.

The operation sequence implemented in the first design consisted of performing arithmetic and `we-` operations in parallel to the execution of multiplications. A total of 33 operations are performed parallel to the multiplications, including the squaring-, addition-, `we-`, `seta`, and `setb` operations. For this reason, 36 clock cycles are sufficient for performing the rest of these operations and the main loop can be executed in this period of time and with the same operation sequence as the one described in the flow chart diagrams of section 5.1.3 (see Figures 5.3, 5.4, 5.5 and 5.6). In contrast to the operation sequence described in the diagrams of section 5.1.3, the operations are performed parallel to the partial multiplications in almost every clock cycle. Consequently, the programs `montk1`, `montk0` and their variations were implemented in the same way but without waiting times between the states of the programs. The diagram in Figure 6.1 shows how this operation sequence has been applied in loops consisting of only 36 clock cycles. This type of operation sequence (with the corresponding interchangeable use of the registers) has been applied to both programs `montk1` and `montk0` and their variations.



Clock cycle	state	Control (HEX)
0	14 (mont)	06004000
1	3	02100000
2	4	07001000
3	5	07100000
4	6	05200000
5	7	02400000
6	8	06008000
7	9	07004000
8	10	04100000
9	11	07020000
10	12	02200000
11	13	01400000
12	14	06010000
13	15	03040000
14	15	00000000
15	16	04080000
16	17	04200000
17	18	03400000
18	19	06010000
19	20	07100000
20	20	00000000
21	21	07008000
22	22	07200000
23	23	08400000
24	24	06004000
25	25	05100000
26	26	04080000
27	27	07010000
28	28	05200000
29	29	0A400000
30	30	06040000
31	31	02080000
32	32	00000040
33	33	07020000
34	34	03200000
35	35	04400000
36 (cycle 0 from next loop)	14 (mont)	06004000
37 (cycle 1 from next loop)	3	02100000
. . .	. . .	. . .

Figure 6.1: Flow chart diagram of `montk0` adapted to the 6-clock-cycle multiplier.

By executing the programs `montk1` and `montk0` in 36 clock cycles, it is assured that the Multiplier will not enter an inactive state in any cycle while performing the loop iteration. This was important to achieve, not only for optimizing the execution time of the implementation but also to avoid possible vulnerabilities to SCA caused by the Multiplier's inactivity (see p. 66 in section 4.16).

The main difference the operation sequence in Figure 6.1 has compared to the operation sequence applied to the ECC implementation with the 9-clock-cycle multiplier (see Figure 5.4), is the fact that squaring, addition and `we-` operations are performed during almost every clock cycle. In the re-designed implementation with the 9-clock-cycle multiplier, these operations are performed at least every 2 clock cycles in order to cause a regular power consumption of the ALU and the registers along the loop execution (see p. 84 in section 5.1.3). This is also achieved for the ECC implementation with the 6-clock-cycle multiplier, since the regularity with which each one of these operations is performed remains constant. For this reason, no anomalies such as big dips or plateaus can be detected along the simulated or measured PT. The boundaries between the slots corresponding to single loop iteration executions are not easily identifiable. Figure 6.2 shows a PT simulated for the re-designed ECC implementation with the 6-clock-cycle multiplier.

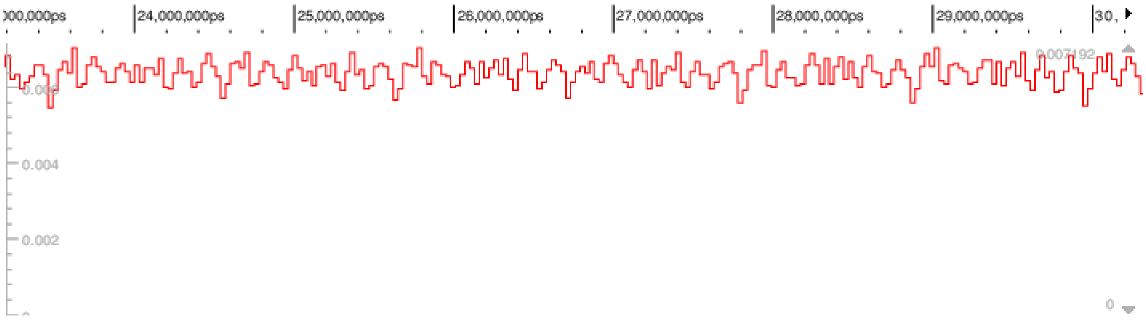


Figure 6.2: Part of a PT simulated for the re-designed ECC implementation with 6-clock-cycle multiplier while executing the kP-operation.
The graph shows the power consumption (in W) over the time (in ps). No big dips or plateaus in the trace can be easily detected and the boundaries between the slots are not easily identifiable.

In contrast to the PT simulated for the re-designed ECC implementation with the 9-clock-cycle multiplier (see Figure 5.11 in p. 91), the PT in Figure 6.2 repeatedly shows small dips. These dips are not easily observable; the difference in the power amplitude between the maximum and minimum values in a slot is not big. Nevertheless it is not excluded that these dips could be helpful for identifying the boundaries between the slots in the PT.

6.1.2 Re-design of the First Loop Iteration

The re-design methodology for the execution of the first loop iteration can also be applied to the ECC implementation with the 6-clock-cycle multiplier. Analogue to the design with the 9-clock-cycle multiplier, the number of clock cycles needed for executing the first loop iteration can be reduced by performing one multiplication less than for the rest of the loop iterations. The minimum time of execution of the first loop is thus 30 clock cycles (5 multiplications · 6 clock cycles). 29 other operations need to be executed in parallel to the multiplications. Figure 6.3 shows a simulated PT at the beginning of the kP-operation.

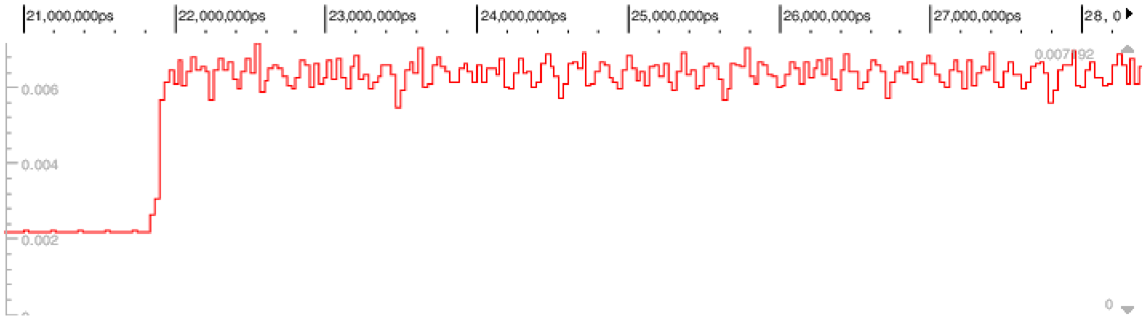


Figure 6.3: Part of a simulated PT at the beginning of the execution of the kP-operation. The graph shows the power consumption (in W) over the time (in ps). The slot of the first loop iteration execution is not easily identifiable.

In contrast to the re-designed ECC implementation with 9-clock-cycle multiplier, the slot corresponding to the first loop iteration does not show an obvious plateau at the beginning (see Figure 5.12 in p. 92). The power consumption is more constant since arithmetic, `we-` and `set_` operations are performed during almost each clock cycle of the loop, just like for the rest of the loop iterations. Nevertheless, the first slot does not reveal information about the key bit being processed, since no multiplications are performed using an operand with value = 1[1]. This provides protection against FSA and other attacks that use initial data execution and corresponding slots as templates when the value of the processed key bit is known.

[1] If the first loop iteration of the implemented algorithm is performed in the same way as the rest of the iterations, several multiplications in the loop are performed with operands with value = 1 as this is the initialized value of register Z1. The effects of performing multiplications with operands with value = 1 are explained in the fifth point of section 4.4, p. 69.

6.1.3 Chip Area and Power Consumption of the new Design

Analogue to the re-design process of the IHP ECC implementation with the 9-clock-cycle multiplier, the modifications made on the IHP ECC implementation with the 6-clock-cycle multiplier implied increases on the design's chip area and average power consumption. Nevertheless the execution time of each key bit processing was reduced from 41 to 36 clock cycles. As a consequence, the energy consumption per kP-operation was reduced by 4.7%. Table 6.1 shows the power, time and energy consumption of the original and the re-designed ECC implementation with the 6-clock-cycle multiplier. The execution time is calculated by the number of clock cycles taken for a complete execution of the kP-operation, with one clock cycle having a period of $T = 0.30$ ns. Changes on the area of the implementation are also shown.

Table 6.1: Area, power, time and energy consumption of the original and re-designed versions of the ECC implementation with the 6-clock-cycle multiplier.

ECC version	**original**	**re-designed**
Power consumption	5.63 mW	6.19 mW
Time of a kP-operation	0.30 ms	0.26 ms
Energy consumption per kP-operation	1.69 μJ	1.61 μJ
Area	0.281117 mm^2	0.294870 mm^2

6.2 Results of the Difference-of-Means Test

The original IHP ECC design implemented with the 6-clock-cycle multiplier was not resistant to a horizontal DPA attack using the difference-of-means test (see the red dotted curve in Figure 6.4). The test delivered 41 key candidates. Four of them had a relative correctness of about 90% (see key candidates 34, 35, 40, 41) and key candidate 1 was extracted with a correctness of 100% (see red dotted curve in Figure 6.4).

The difference-of-means test was again performed on the implementation after its re-design. The test was performed in the same way as described in sections 4.2 and 5.3. 230 slots were used for the test and 36 key candidates were obtained (see the black curve in Figure 6.4).

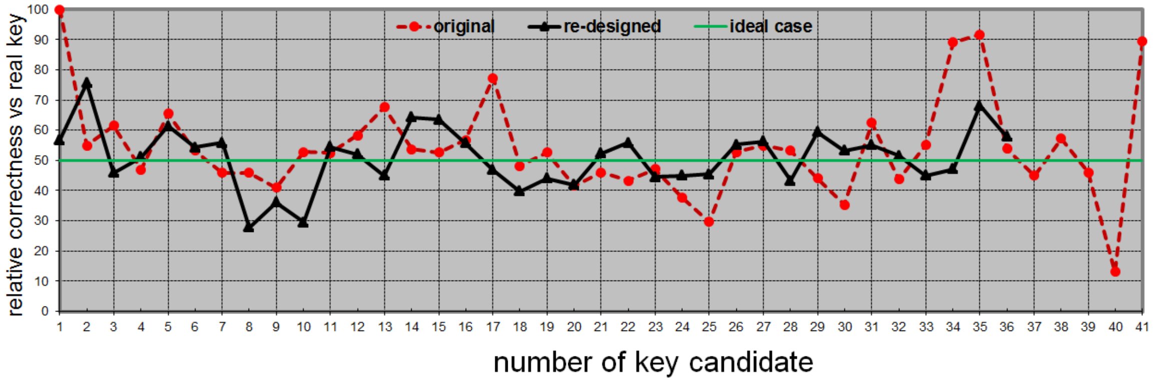


Figure 6.4: Relative correctness for the original ECC implementation with 6-clock-cycle multiplier (red dotted curve) and for its re-designed version (black curve).

The re-designed ECC implementation delivers 36 key candidates. These are five candidates less than the 41 delivered by the original implementation. None of the 36 new key candidates has a relative correctness higher than 76%. So no point can be pointed out as a strong SCA leakage source.

It can be concluded that the re-design made on a second version of the ECC implementation has made it more resistant against the horizontal DPA attack using the difference-of-means test.

7 Conclusions

The work described in this thesis consisted of analysing the implementation details of an IHP ECC design, which was not resistant to a horizontal DPA attack performed using the difference-of-means test. Subsequently, this ECC implementation was re-designed and its vulnerability against SCA was successfully reduced. The re-design ideas were applied to two ECC implementations with the same structure but with a different type of Multiplier each. This re-design methodology can be used for all others implementations of the Montgomery kP-algorithm which are based on a bus architecture and consist of an ALU and a multiplier that perform their operations independently of each other.

In order to understand the reasons why the original IHP ECC implementation was not resistant against the difference-of-means test, the implementation's system architecture and source code were analysed in detail. This helped to understand, which entities were part of this design and how they were connected to each other. Subsequently, the activities performed by all entities during the execution of the kP-algorithm were analysed. Three major observations were made:

- The implementation of the Montgomery kP-algorithm was not balanced.
- The key-bit value dependent use of the registers in the IHP implementation of this algorithm is a strong SCA leakage source.
- The internal power consumption of the FUs in the design may lead to SCA leakage as well.

In addition to this, the executions of unnecessary instructions were identified.

The ECC implementation was carefully re-designed considering the observations mentioned above. The operation sequence for each bit processing was improved. A more regular power consumption was achieved for the performance of all the bit processing operations and each bit processing time was optimized. The number of clock cycles needed for processing one key bit was reduced from 57 to 54 and from 41 to 36 for the 9-clock-cycle and the 6-clock-cycle implementations respectively. A faster execution time was achieved and at the same time, the ECC implementations became more robust against SCA.

Figure 7.1 shows a comparison of the difference-of-means test results for the original implementation with the 9-clock-cycle multiplier (see the red dotted curve) and for its re-design (see the black solid curve).

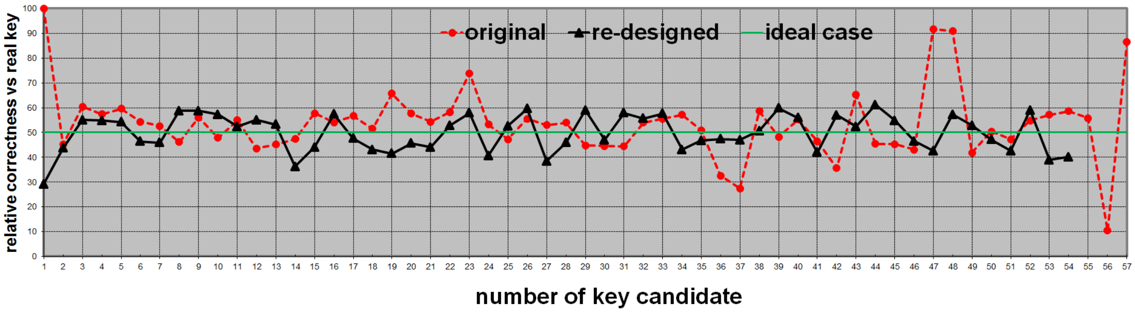


Figure 7.1: Relative correctness curves for the original ECC implementation with 9-clock-cycle multiplier (red dotted curve) and the re-designed version of this implementation (black curve).

In the original implementation five operations leaked so much information about the private key that it could be extracted with a correctness of over 90% (see Figure 7.1). One of the key candidates was even extracted with 100% correctness. The re-designed ECC implementation delivers three key candidates less than the original one, which reduces the probabilities of extracting the private key successfully with this test. The new implementation does not deliver any point with a relevant key extraction correctness.
The same re-design ideas, i.e. the same re-design methodology, applied to the ECC design with 6-clock-cycle multiplier also improved the resistance of this design against the difference-of-means test. The original design delivered four key candidates with a relative correctness of about 90% and one with a correctness of 100%. The re-designed implementation does not deliver any point with a key extraction correctness higher than 75% and it delivers five key candidates less than the original one (see Figure 6.4 in section 6.2).

The improved resistance of the ECC design against PA was achieved by adding two new registers to the system architecture and increasing the number of operations performed while processing a key bit. This led to a small increase in the chip area needed for the design and its average power consumption. Nevertheless the energy consumption caused by performing a complete kP-operation remained almost the same for the implementation with the 9-clock-cycle multiplier and was even reduced for the implementation with the 6-clock-cycle multiplier. This was achieved due to the shorter execution time needed for performing the complete kP-operation. Tables 7.1 and 7.2 summarize the results obtained for both re-designed ECC implementations.

Table 7.1: Comparison of the results of the original ECC implementation with the 9-clock-cycle multiplier and its re-designed version.

ECC version	**original**	**re-designed**
Clock cycles per key bit-processing	57	54 (reduced by 5%)
Clock cycles per kP-operation	13606	12904 (reduced by 5%)
Time of a kP-operation	$13606 \cdot 30$ ns ≈ 0.41 ms	$12904 \cdot 30$ ns ≈ 0.39 ms (reduced by 5%)
Power consumption	4.90 mW	5.40 mW (increased by 10%)
Energy consumption per kP-operation	2 μJ	2.09 μJ (increased by 4.5%)
Area	0.260258 mm^2	0.274844 mm^2 (increased to 5.6%)

Table 7.2: Comparison of the results of the original ECC implementation with the 6-clock-cycle multiplier and its re-designed version.

ECC version	**original**	**re-designed**
Clock cycles per key-processing	41	36 (reduced by 12%)
Clock cycles per kP-operation	9910	8749 (reduced by 11.7%)
Time of a kP-operation	$9910 \cdot 30$ ns ≈ 0.30 ms	$8749 \cdot 30$ ns ≈ 0.26 ms (reduced by 11.7%)
Power consumption	5.63 mW	6.19 mW (increased by 9%)
Energy consumption per kP-operation	1.69 μJ	1.61 μJ (reduced by 4.7%)
Area	0.281117 mm^2	0.294870 mm^2 (increased by 4.9%)

It was shown in section 5.4 that the re-designed implementation is still vulnerable against ComPA attacks. Therefore, further countermeasures against DPA should be considered for achieving a more secure ECC implementation.

Appendix A Acronyms

ALU	Arithmetic Logic Unit
CMOS	Complementary Metal-Oxide-Semiconductor
ComPA	Comparative Power Analysis
DPA	Differential Power Analysis
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
FI	Fault Injection
FS	Fault Sensitivity
FSA	Fault Sensitivity Analysis
FU	Functional Unit
GF	Galois Fields
IC	Integrated Circuit
IHP	Innovations for High Performance Microelectronics
LSB	Least Significant Bit
MSB	Most Significant Bit
PA	Power Analysis
PT	Power Trace
SCA	Side-Channel Analysis
SPA	Simple Power Analysis
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WSNs	Wireless Sensor Networks

Appendix B Flow Diagrams with Power Traces

The figures in this appendix show the flow diagrams and PTs for all slots of the kP-operation simulated using the EC point $P1 = (x_1, y_1)$ (see section 4.2) and a small key $k = 2cc$.

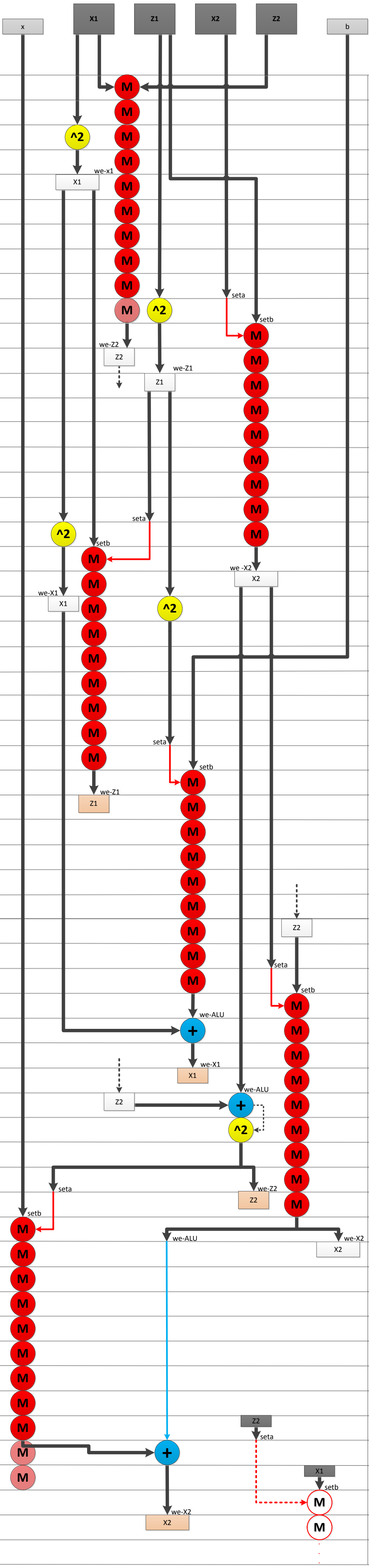
The most significant bit of k is processed in the initialization part of the Montgomery kP-algorithm and is not shown here. To illustrate the leakage sources, the power consumption curves of the ECC design and of its single entities are shown in the same flow diagram as traces. This way, it is possible to observe the power consumption of each entity during the individual clock cycles.

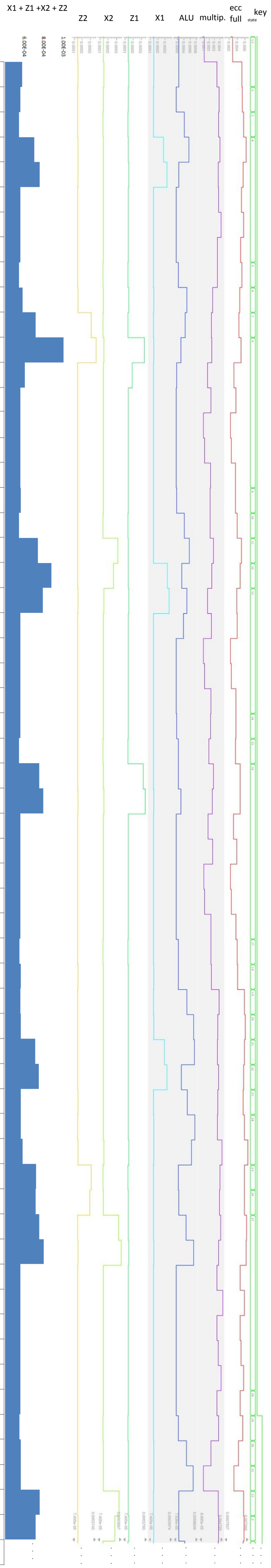
The power consumption of the following entities is shown on the right side of each diagram:

- Complete ECC design
- Multiplier
- ALU
- X1 register
- Z1 register
- X2 register
- Z2 register
- The sum of the registers X1, Z1, X2, Z2

A total of 9 diagrams display how the 9 bits of the 10 bit long key are processed.

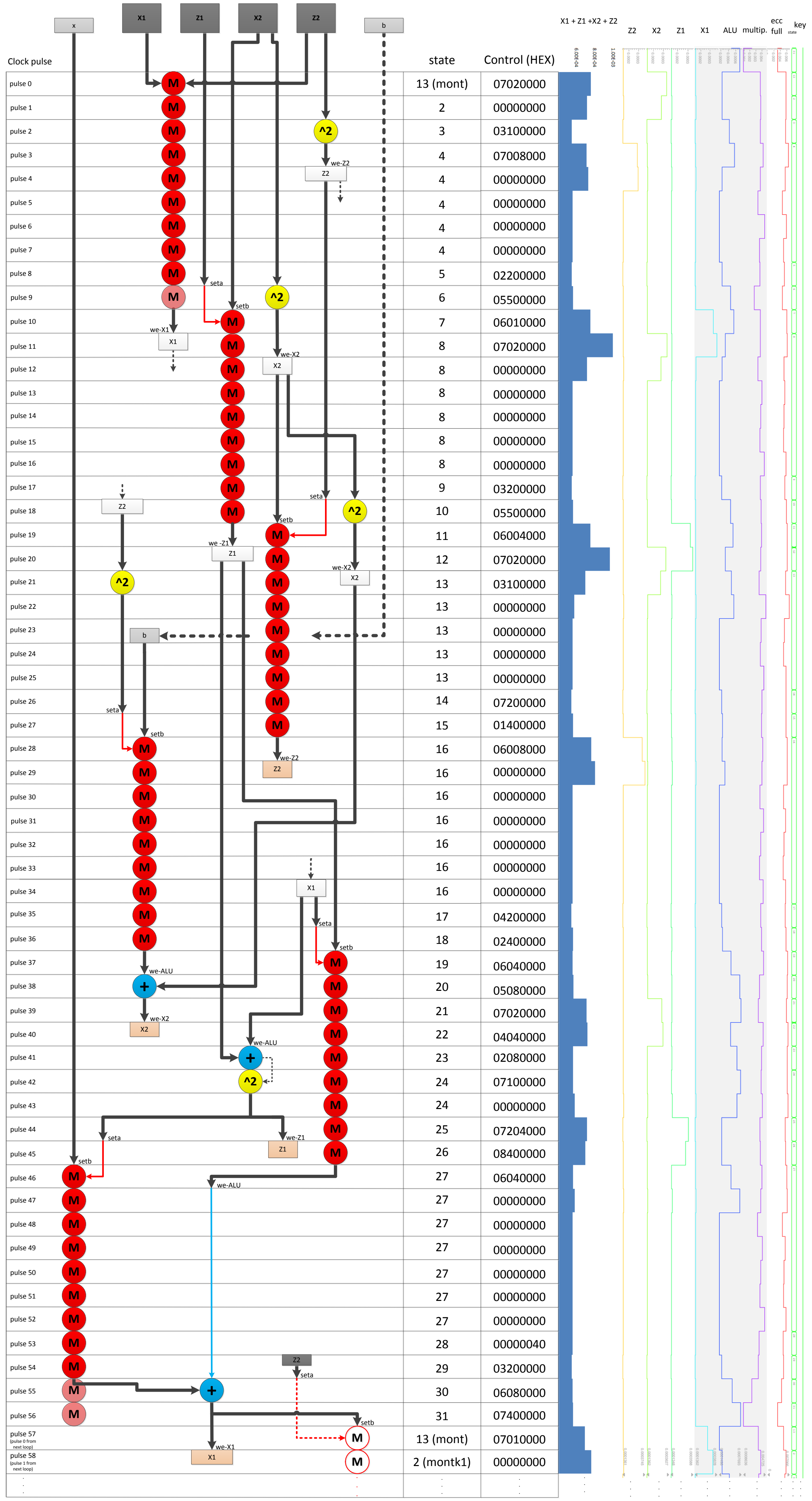
Montk0 (1)





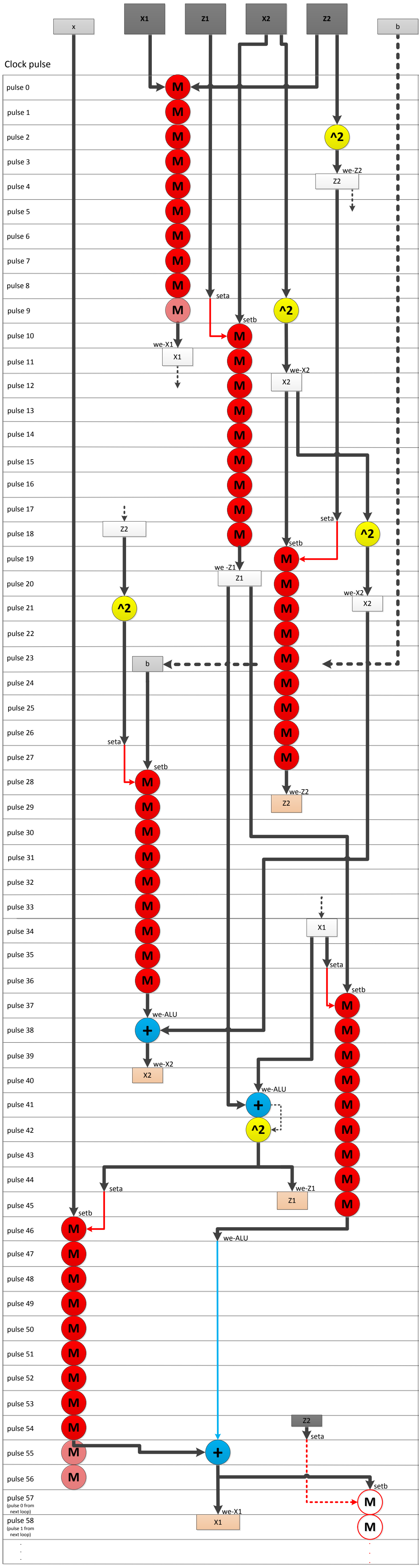
Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	00000000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07020000
pulse 58 (pulse 1 from next loop)	2 (montk1)	00000000
⋮	⋮	⋮

montk1 (2)

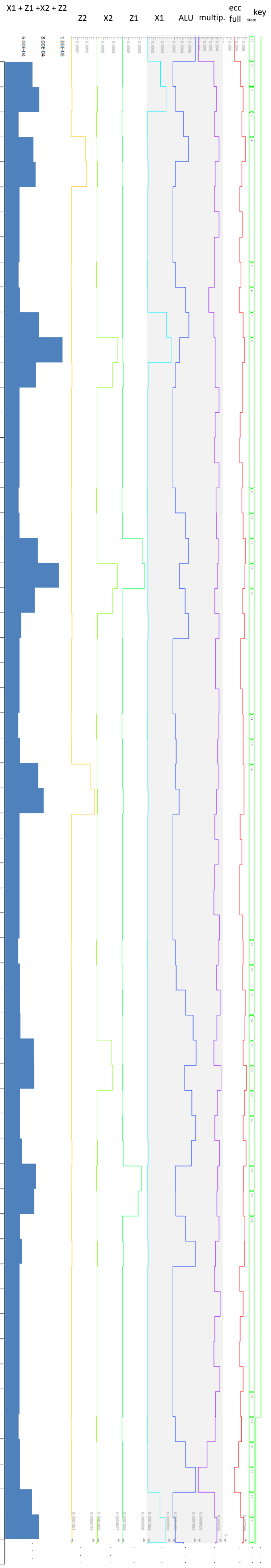


Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07020000
pulse 1	2	00000000
pulse 2	3	03100000
pulse 3	4	07008000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	02200000
pulse 9	6	05500000
pulse 10	7	06010000
pulse 11	8	07020000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	03200000
pulse 18	10	05500000
pulse 19	11	06004000
pulse 20	12	07020000
pulse 21	13	03100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06008000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	04200000
pulse 36	18	02400000
pulse 37	19	06040000
pulse 38	20	05080000
pulse 39	21	07020000
pulse 40	22	04040000
pulse 41	23	02080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07204000
pulse 45	26	08400000
pulse 46	27	06040000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	07400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07010000
pulse 58 (pulse 1 from next loop)	2 (montk1)	00000000
. . .	. . .	. . .

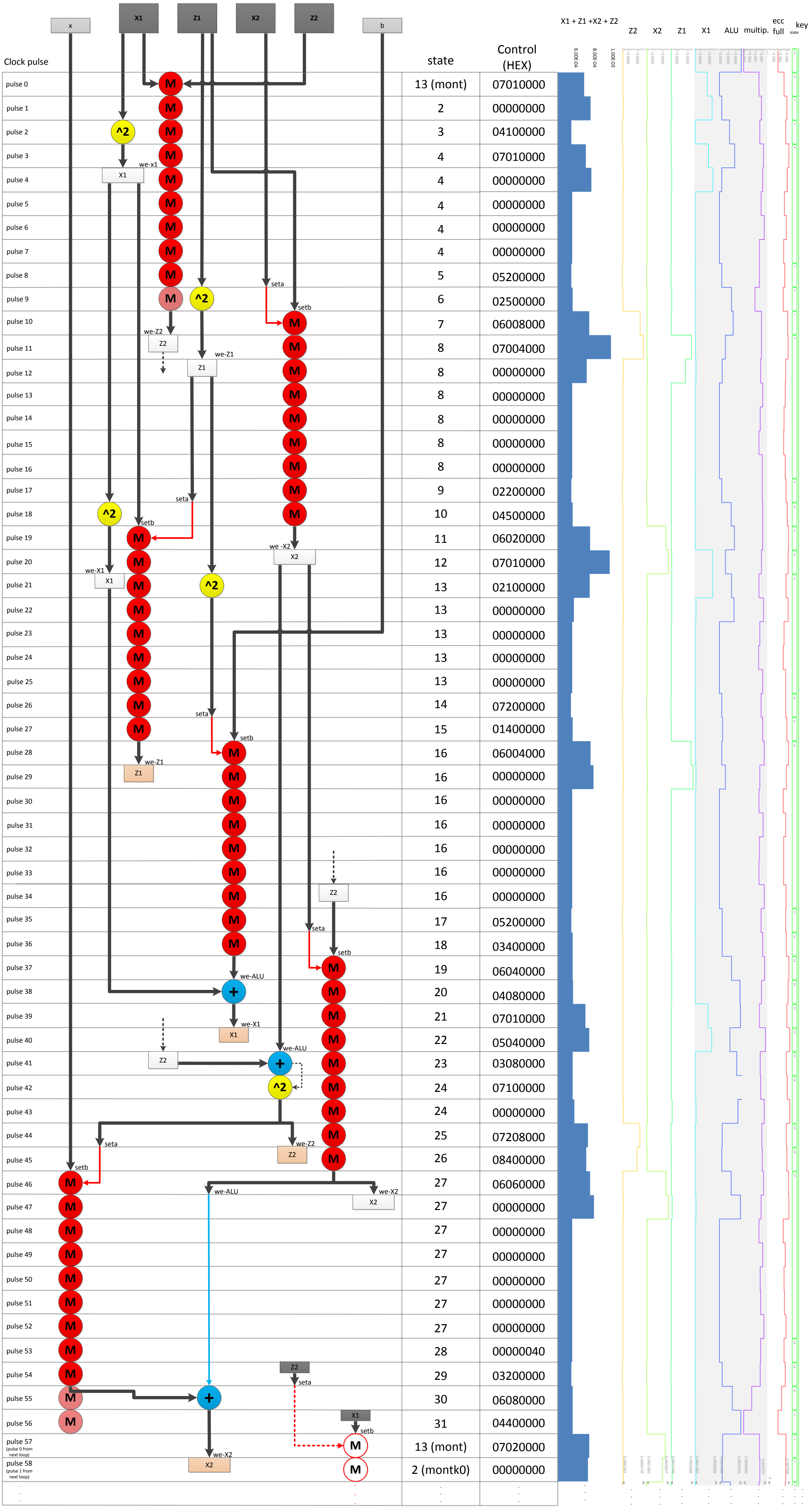
montk1 (3)



Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07010000
pulse 1	2	00000000
pulse 2	3	03100000
pulse 3	4	07008000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	02200000
pulse 9	6	05500000
pulse 10	7	06010000
pulse 11	8	07020000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	03200000
pulse 18	10	05500000
pulse 19	11	06004000
pulse 20	12	07020000
pulse 21	13	03100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06008000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	04200000
pulse 36	18	02400000
pulse 37	19	06040000
pulse 38	20	05080000
pulse 39	21	07020000
pulse 40	22	04040000
pulse 41	23	02080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07204000
pulse 45	26	08400000
pulse 46	27	06040000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	07400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07010000
pulse 58 (pulse 1 from next loop)	2 (montk0)	00000000
⋮	⋮	⋮

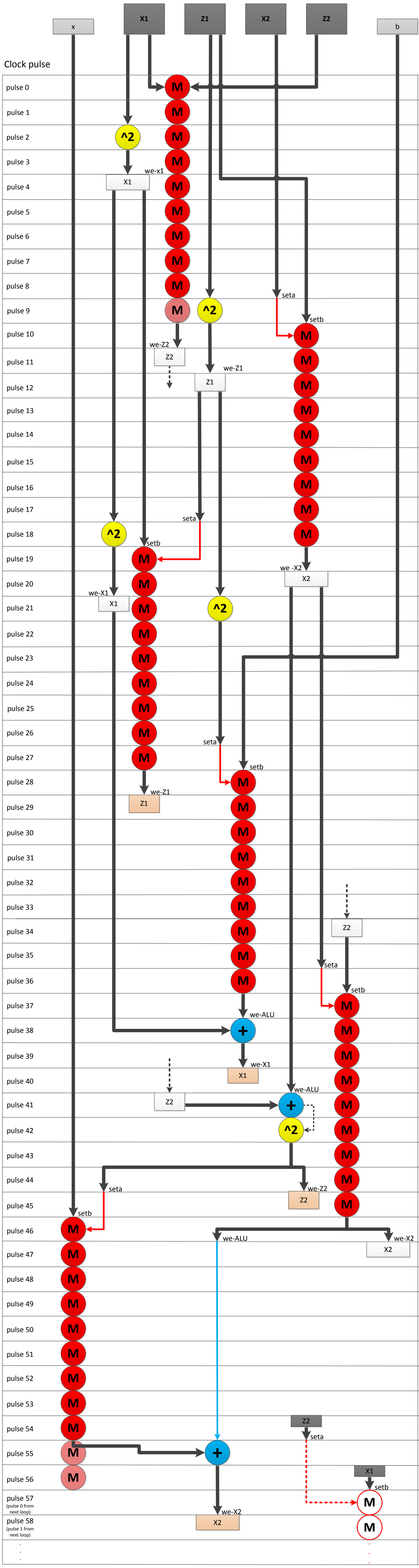


Montk0 (4)

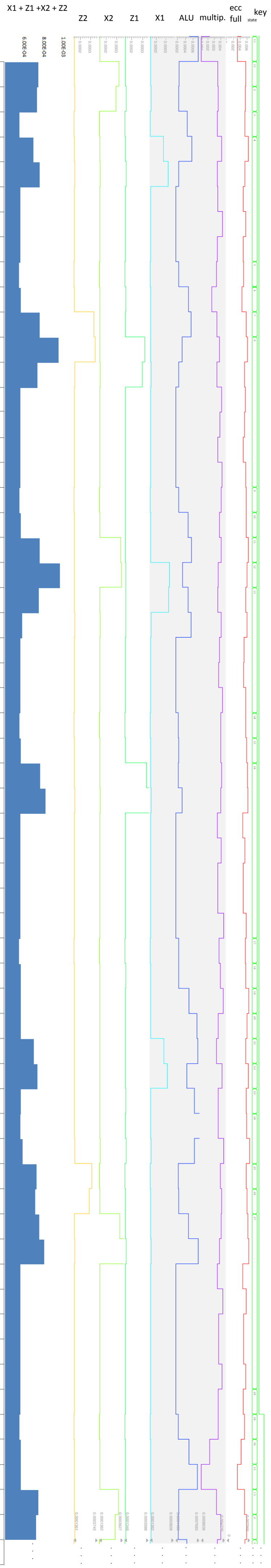


Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07010000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07020000
pulse 58 (pulse 1 from next loop)	2 (montk0)	00000000
. . .	. . .	. . .

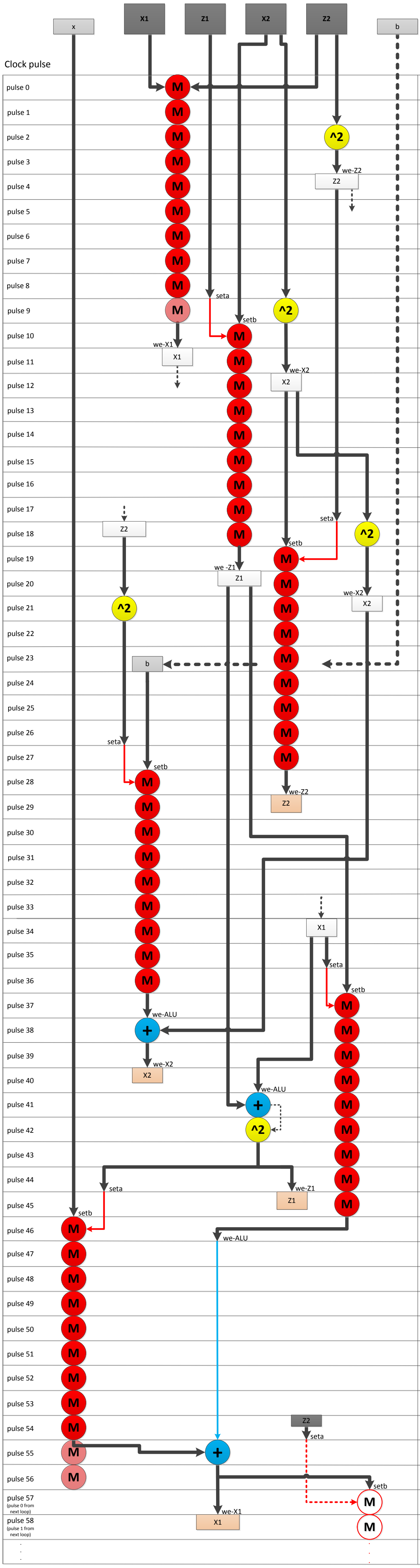
Montk0 (5)



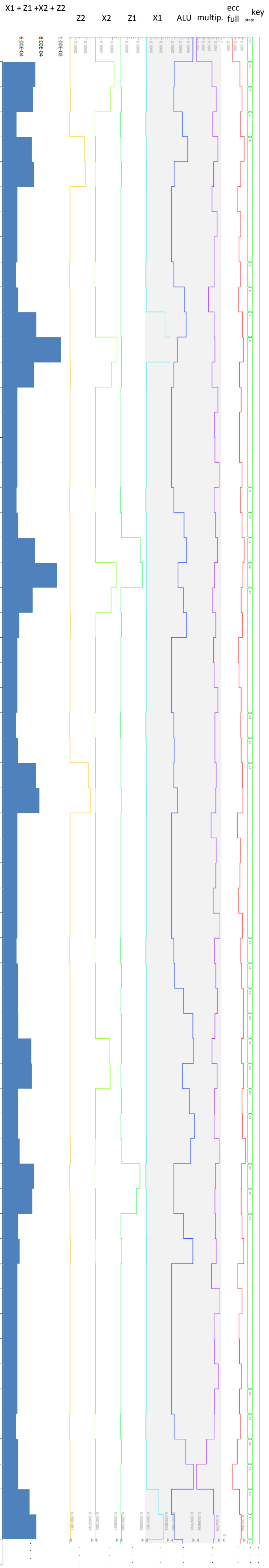
Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07020000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07020000
pulse 58 (pulse 1 from next loop)	2 (montk1)	00000000
. . .	. . .	. . .



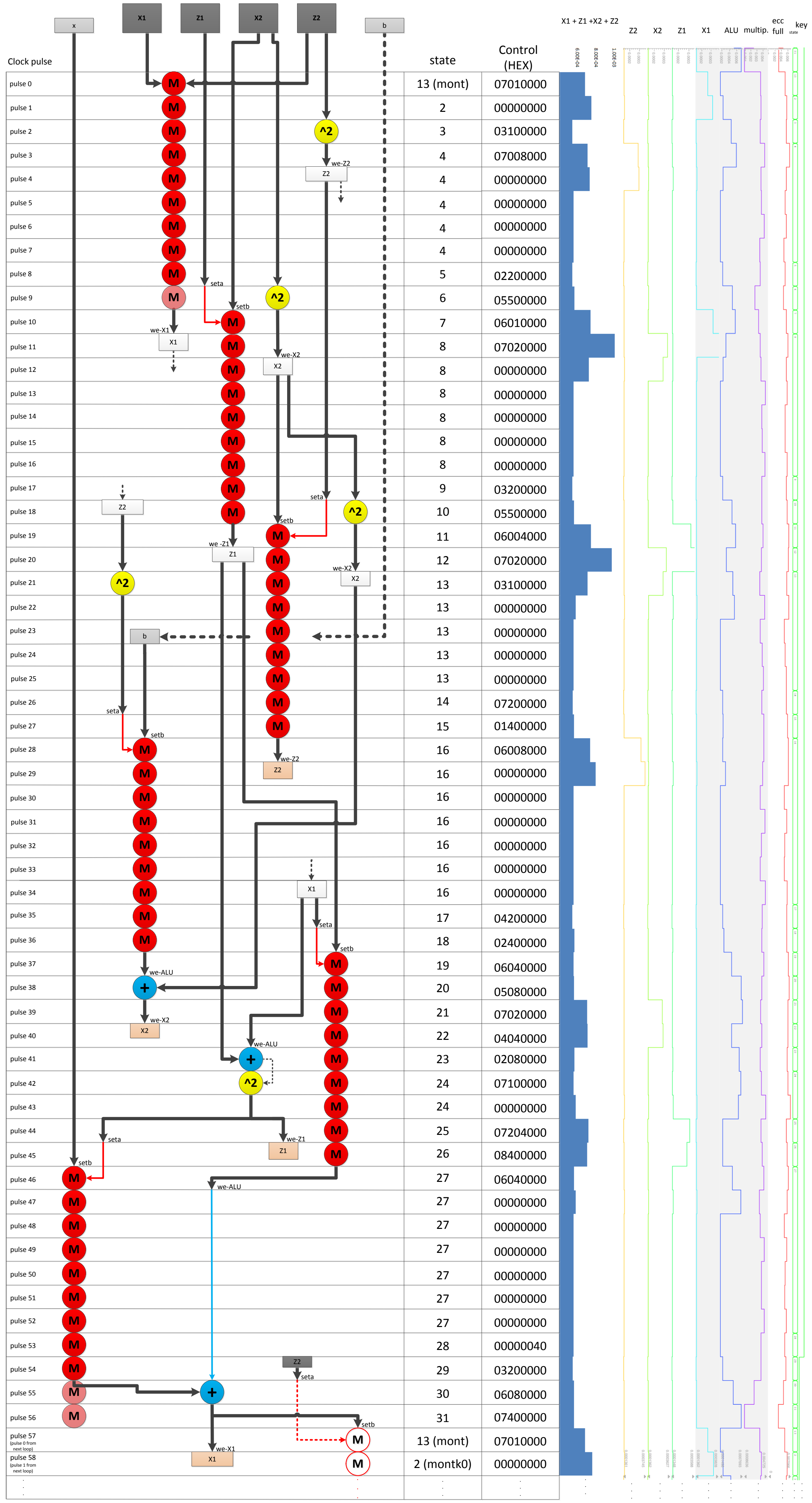
montk1 (6)



Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07020000
pulse 1	2	00000000
pulse 2	3	03100000
pulse 3	4	07008000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	02200000
pulse 9	6	05500000
pulse 10	7	06010000
pulse 11	8	07020000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	03200000
pulse 18	10	05500000
pulse 19	11	06004000
pulse 20	12	07020000
pulse 21	13	03100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06008000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	04200000
pulse 36	18	02400000
pulse 37	19	06040000
pulse 38	20	05080000
pulse 39	21	07020000
pulse 40	22	04040000
pulse 41	23	02080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07204000
pulse 45	26	08400000
pulse 46	27	06040000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	07400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07010000
pulse 58 (pulse 1 from next loop)	2 (montk1)	00000000
. . .	. . .	. . .

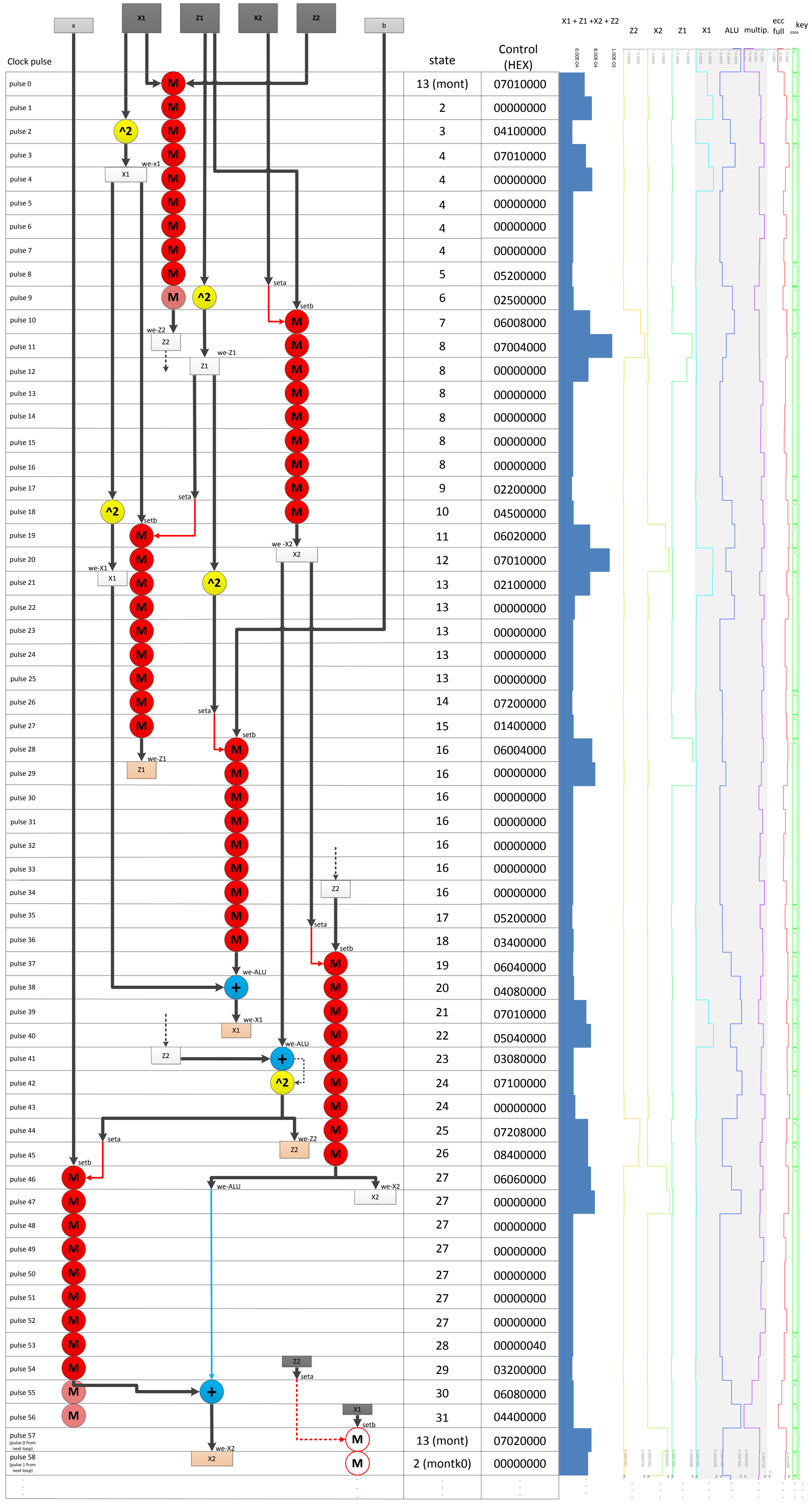


montk1 (7)



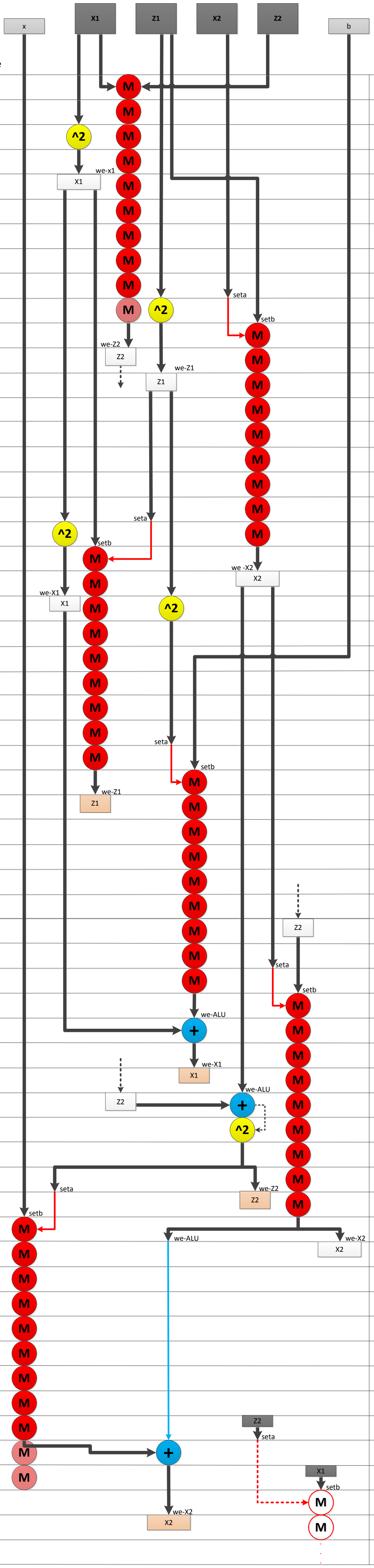
Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07010000
pulse 1	2	00000000
pulse 2	3	03100000
pulse 3	4	07008000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	02200000
pulse 9	6	05500000
pulse 10	7	06010000
pulse 11	8	07020000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	03200000
pulse 18	10	05500000
pulse 19	11	06004000
pulse 20	12	07020000
pulse 21	13	03100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06008000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	04200000
pulse 36	18	02400000
pulse 37	19	06040000
pulse 38	20	05080000
pulse 39	21	07020000
pulse 40	22	04040000
pulse 41	23	02080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07204000
pulse 45	26	08400000
pulse 46	27	06040000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	07400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07010000
pulse 58 (pulse 1 from next loop)	2 (montk0)	00000000
⋮	⋮	⋮

Montk0 (8)

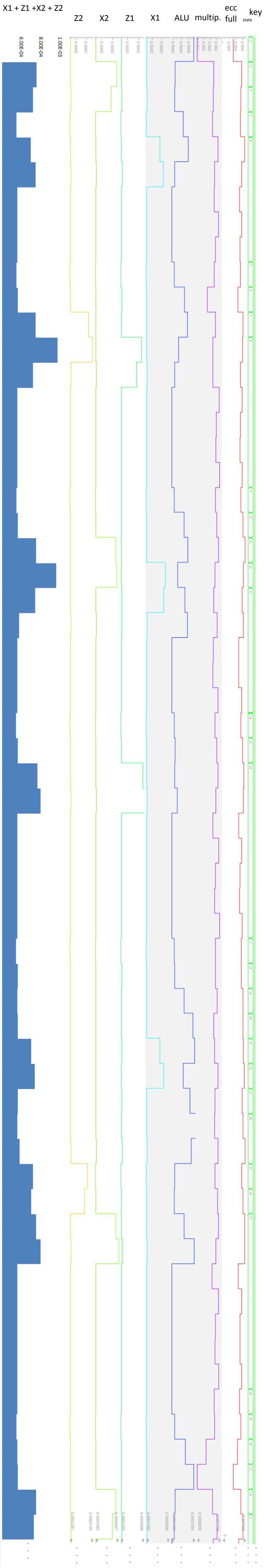


Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07010000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	13 (mont)	07020000
pulse 58 (pulse 1 from next loop)	2 (montk0)	00000000
. . .	. . .	. . .

Montk0 (9)



Clock pulse	state	Control (HEX)
pulse 0	13 (mont)	07020000
pulse 1	2	00000000
pulse 2	3	04100000
pulse 3	4	07010000
pulse 4	4	00000000
pulse 5	4	00000000
pulse 6	4	00000000
pulse 7	4	00000000
pulse 8	5	05200000
pulse 9	6	02500000
pulse 10	7	06008000
pulse 11	8	07004000
pulse 12	8	00000000
pulse 13	8	00000000
pulse 14	8	00000000
pulse 15	8	00000000
pulse 16	8	00000000
pulse 17	9	02200000
pulse 18	10	04500000
pulse 19	11	06020000
pulse 20	12	07010000
pulse 21	13	02100000
pulse 22	13	00000000
pulse 23	13	00000000
pulse 24	13	00000000
pulse 25	13	00000000
pulse 26	14	07200000
pulse 27	15	01400000
pulse 28	16	06004000
pulse 29	16	00000000
pulse 30	16	00000000
pulse 31	16	00000000
pulse 32	16	00000000
pulse 33	16	00000000
pulse 34	16	00000000
pulse 35	17	05200000
pulse 36	18	03400000
pulse 37	19	06040000
pulse 38	20	04080000
pulse 39	21	07010000
pulse 40	22	05040000
pulse 41	23	03080000
pulse 42	24	07100000
pulse 43	24	00000000
pulse 44	25	07208000
pulse 45	26	08400000
pulse 46	27	06060000
pulse 47	27	00000000
pulse 48	27	00000000
pulse 49	27	00000000
pulse 50	27	00000000
pulse 51	27	00000000
pulse 52	27	00000000
pulse 53	28	00000040
pulse 54	29	03200000
pulse 55	30	06080000
pulse 56	31	04400000
pulse 57 (pulse 0 from next loop)	14 (mont)	07020000
pulse 58 (pulse 1 from next loop)	1 (montkpost)	00000000
. . .	. . .	. . .

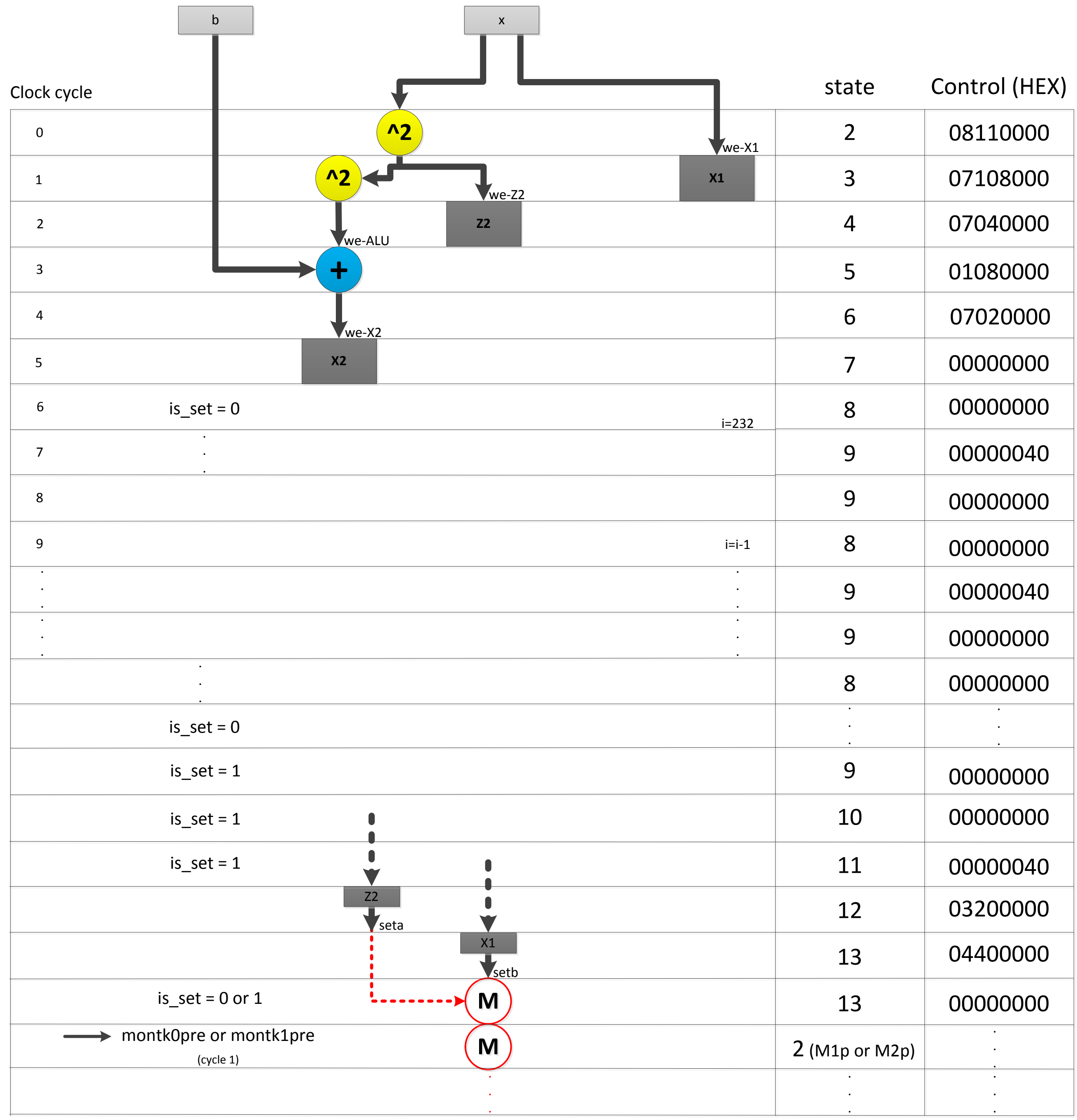


Appendix C Re-designed Flow Charts

The figures in this appendix show flow chart diagrams of all re-designed programs for the implementation of the Montgomery kP-algorithm.

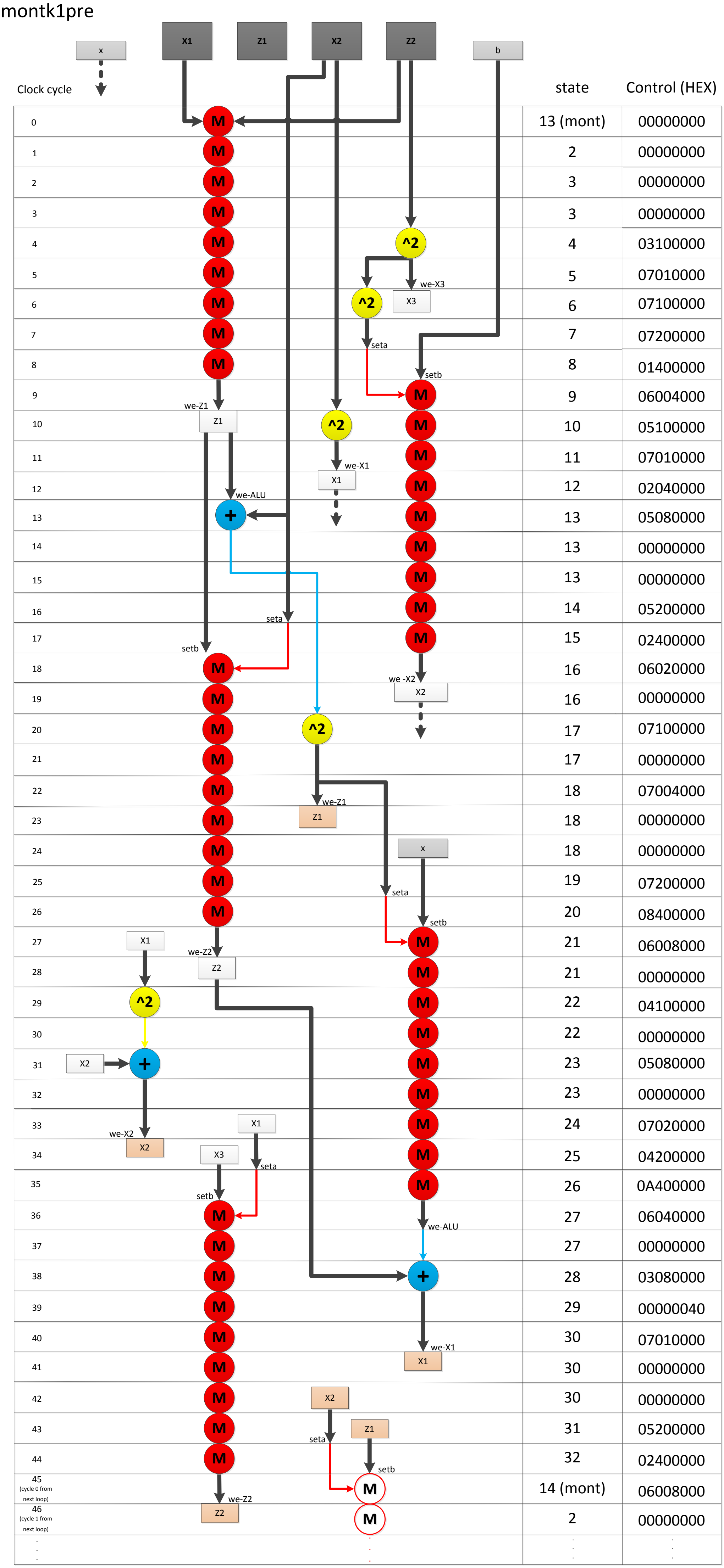
The initialization phase of the algorithm, executed by the program `mont`, is described in the diagram titled **Start: mont**. The programs `montk1pre` and `montk0pre`, which execute the first loop iteration, are described in the diagrams titled **montk1pre** and **montk0pre** respectively. The programs `montk1` and `montk0` and their variations are described in the diagrams titled **montk1**, **montk0**, **montk1 (variant)** and **montk0 (variant)**.

Start: mont



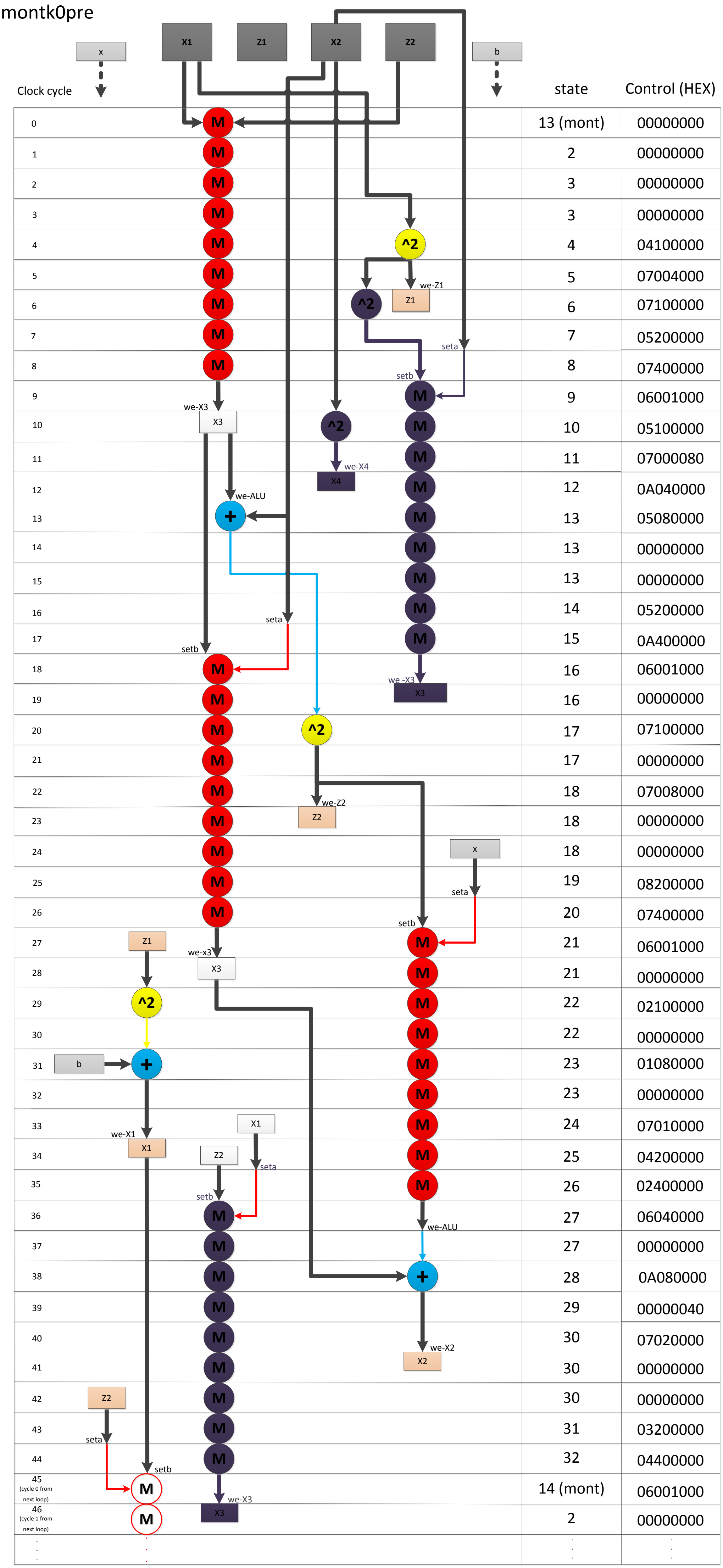
Clock cycle		state	Control (HEX)
0		2	08110000
1		3	07108000
2		4	07040000
3		5	01080000
4		6	07020000
5		7	00000000
6	is_set = 0 i=232	8	00000000
7	. . .	9	00000040
8		9	00000000
9	i=i-1	8	00000000
. . .	. . .	9	00000040
. . .	. . .	9	00000000
	. . .	8	00000000
	is_set = 0	. . .	. . .
	is_set = 1	9	00000000
	is_set = 1	10	00000000
	is_set = 1	11	00000040
		12	03200000
		13	04400000
	is_set = 0 or 1	13	00000000
	→ montk0pre or montk1pre (cycle 1)	2 (M1p or M2p)	. . .
	. . .	. . .	. . .

montk1pre



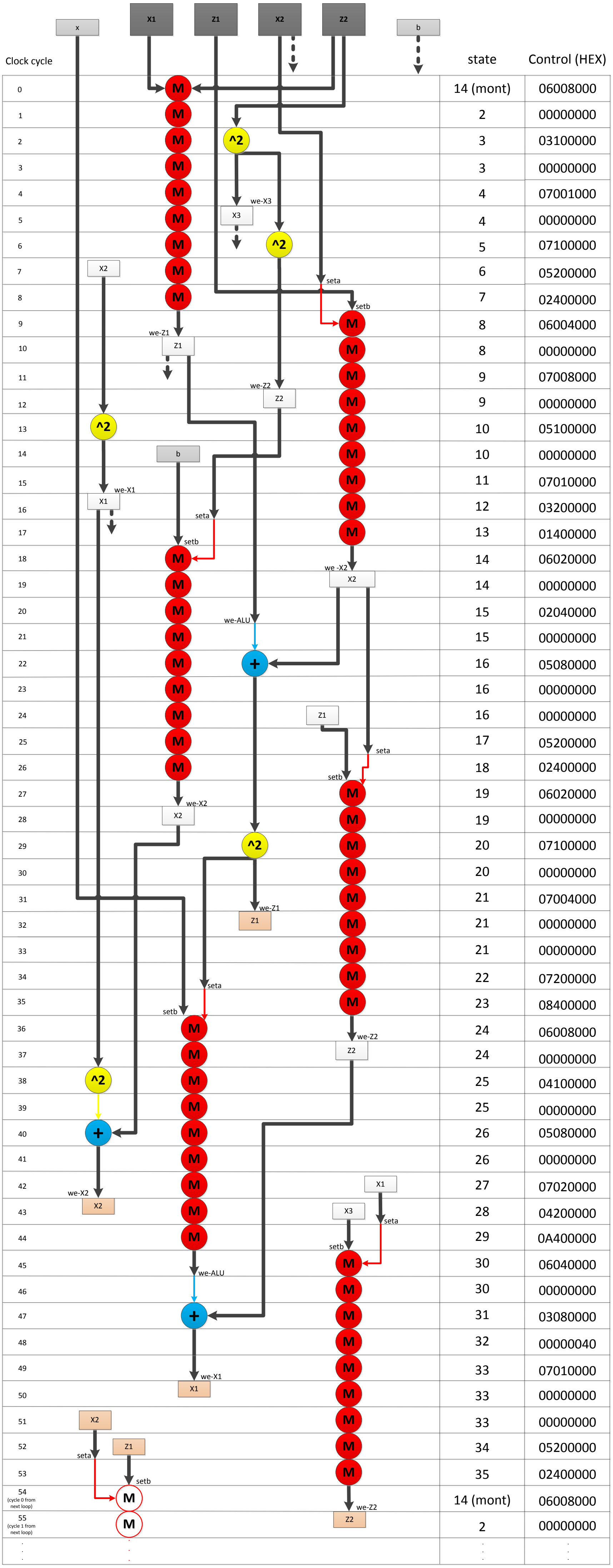
Clock cycle	state	Control (HEX)
0	13 (mont)	00000000
1	2	00000000
2	3	00000000
3	3	00000000
4	4	03100000
5	5	07010000
6	6	07100000
7	7	07200000
8	8	01400000
9	9	06004000
10	10	05100000
11	11	07010000
12	12	02040000
13	13	05080000
14	13	00000000
15	13	00000000
16	14	05200000
17	15	02400000
18	16	06020000
19	16	00000000
20	17	07100000
21	17	00000000
22	18	07004000
23	18	00000000
24	18	00000000
25	19	07200000
26	20	08400000
27	21	06008000
28	21	00000000
29	22	04100000
30	22	00000000
31	23	05080000
32	23	00000000
33	24	07020000
34	25	04200000
35	26	0A400000
36	27	06040000
37	27	00000000
38	28	03080000
39	29	00000040
40	30	07010000
41	30	00000000
42	30	00000000
43	31	05200000
44	32	02400000
45 (cycle 0 from next loop)	14 (mont)	06008000
46 (cycle 1 from next loop)	2	00000000
. . .	. . .	. . .

montk0pre



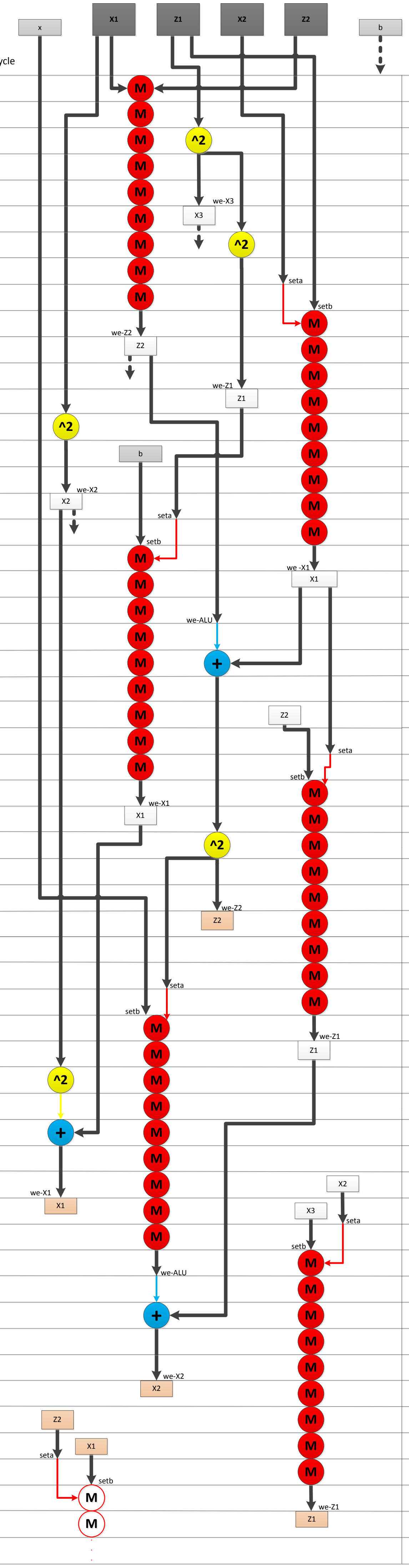
Clock cycle	state	Control (HEX)
0	13 (mont)	00000000
1	2	00000000
2	3	00000000
3	3	00000000
4	4	04100000
5	5	07004000
6	6	07100000
7	7	05200000
8	8	07400000
9	9	06001000
10	10	05100000
11	11	07000080
12	12	0A040000
13	13	05080000
14	13	00000000
15	13	00000000
16	14	05200000
17	15	0A400000
18	16	06001000
19	16	00000000
20	17	07100000
21	17	00000000
22	18	07008000
23	18	00000000
24	18	00000000
25	19	08200000
26	20	07400000
27	21	06001000
28	21	00000000
29	22	02100000
30	22	00000000
31	23	01080000
32	23	00000000
33	24	07010000
34	25	04200000
35	26	02400000
36	27	06040000
37	27	00000000
38	28	0A080000
39	29	00000040
40	30	07020000
41	30	00000000
42	30	00000000
43	31	03200000
44	32	04400000
45 (cycle 0 from next loop)	14 (mont)	06001000
46 (cycle 1 from next loop)	2	00000000
. . .	. . .	. . .

montk1



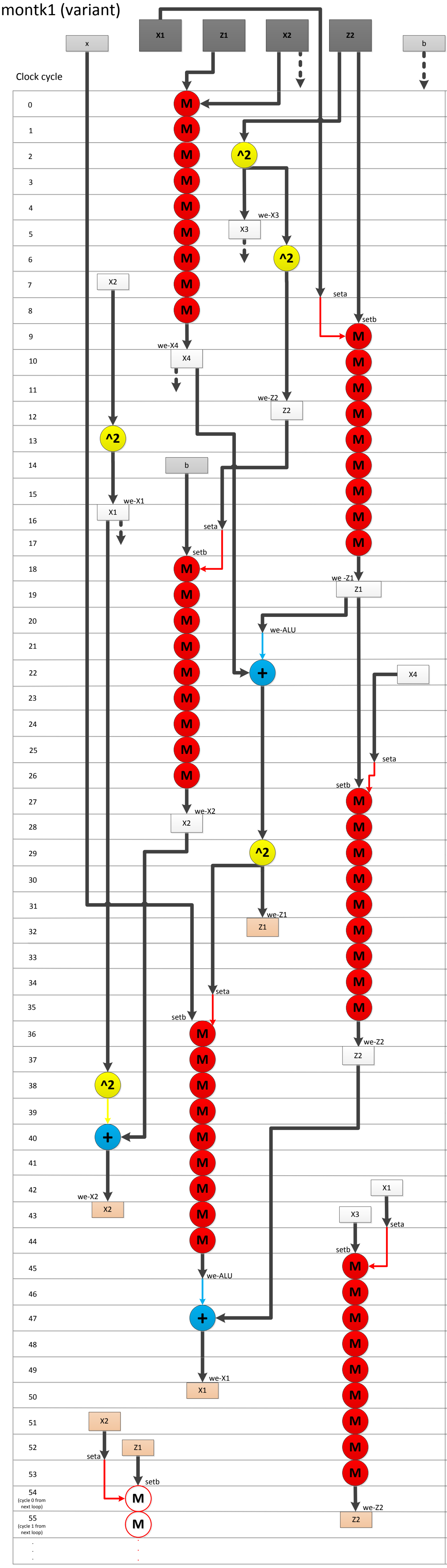
Clock cycle	state	Control (HEX)
0	14 (mont)	06008000
1	2	00000000
2	3	03100000
3	3	00000000
4	4	07001000
5	4	00000000
6	5	07100000
7	6	05200000
8	7	02400000
9	8	06004000
10	8	00000000
11	9	07008000
12	9	00000000
13	10	05100000
14	10	00000000
15	11	07010000
16	12	03200000
17	13	01400000
18	14	06020000
19	14	00000000
20	15	02040000
21	15	00000000
22	16	05080000
23	16	00000000
24	16	00000000
25	17	05200000
26	18	02400000
27	19	06020000
28	19	00000000
29	20	07100000
30	20	00000000
31	21	07004000
32	21	00000000
33	21	00000000
34	22	07200000
35	23	08400000
36	24	06008000
37	24	00000000
38	25	04100000
39	25	00000000
40	26	05080000
41	26	00000000
42	27	07020000
43	28	04200000
44	29	0A400000
45	30	06040000
46	30	00000000
47	31	03080000
48	32	00000040
49	33	07010000
50	33	00000000
51	33	00000000
52	34	05200000
53	35	02400000
54 (cycle 0 from next loop)	14 (mont)	06008000
55 (cycle 1 from next loop)	2	00000000
. . .	. . .	. . .

montk0



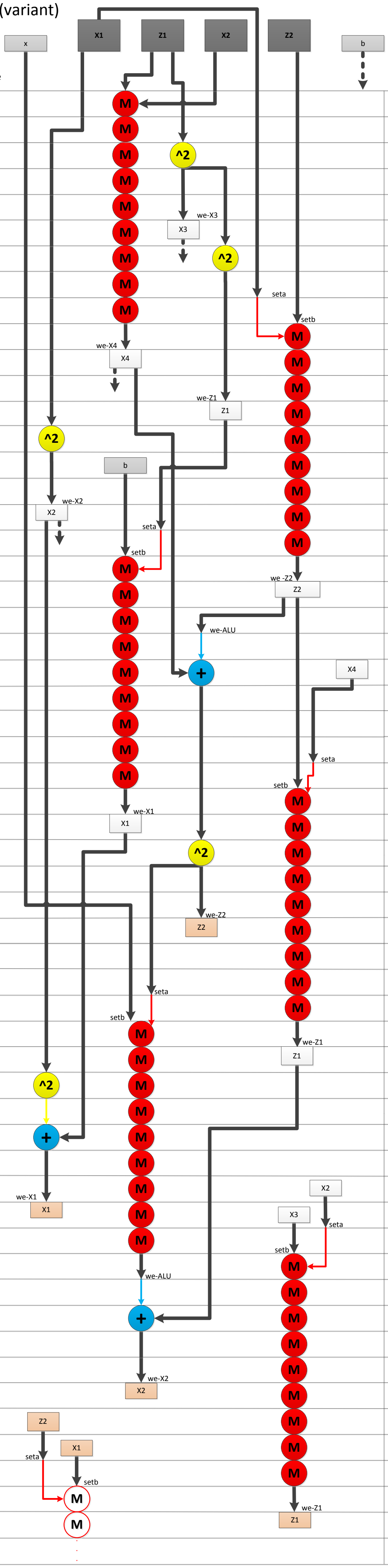
Clock cycle	state	Control (HEX)
0	14 (mont)	06004000
1	2	00000000
2	3	02100000
3	3	00000000
4	4	07001000
5	4	00000000
6	5	07100000
7	6	05200000
8	7	02400000
9	8	06008000
10	8	00000000
11	9	07004000
12	9	00000000
13	10	04100000
14	10	00000000
15	11	07020000
16	12	02200000
17	13	01400000
18	14	06010000
19	14	00000000
20	15	03040000
21	15	00000000
22	16	04080000
23	16	00000000
24	16	00000000
25	17	04200000
26	18	03400000
27	19	06010000
28	19	00000000
29	20	07100000
30	20	00000000
31	21	07008000
32	21	00000000
33	21	00000000
34	22	07200000
35	23	08400000
36	24	06004000
37	24	00000000
38	25	05100000
39	25	00000000
40	26	04080000
41	26	00000000
42	27	07010000
43	28	05200000
44	29	0A400000
45	30	06040000
46	30	00000000
47	31	02080000
48	32	00000040
49	33	07020000
50	33	00000000
51	33	00000000
52	34	03200000
53	35	04400000
54 (cycle 0 from next loop)	14 (mont)	06004000
55 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

montk1 (variant)



Clock cycle	state	Control (HEX)
0	14 (mont)	06008000
1	2	00000000
2	3	03100000
3	3	00000000
4	4	07001000
5	4	00000000
6	5	07100000
7	6	04200000
8	7	03400000
9	8	06000080
10	8	00000000
11	9	07008000
12	9	00000000
13	10	05100000
14	10	00000000
15	11	07010000
16	12	03200000
17	13	01400000
18	14	06004000
19	14	00000000
20	15	02040000
21	15	00000000
22	16	0B080000
23	16	00000000
24	16	00000000
25	17	0B200000
26	18	02400000
27	19	06020000
28	19	00000000
29	20	07100000
30	20	00000000
31	21	07004000
32	21	00000000
33	21	00000000
34	22	07200000
35	23	08400000
36	24	06008000
37	24	00000000
38	25	04100000
39	25	00000000
40	26	05080000
41	26	00000000
42	27	07020000
43	28	04200000
44	29	0A400000
45	30	06040000
46	30	00000000
47	31	03080000
48	32	00000040
49	33	07010000
50	33	00000000
51	33	00000000
52	34	05200000
53	35	02400000
54 (cycle 0 from next loop)	14 (mont)	06008000
55 (cycle 1 from next loop)	2	00000000
. . .	. . .	. . .

montk0 (variant)



Clock cycle	state	Control (HEX)
0	14 (mont)	06008000
1	2	00000000
2	3	02100000
3	3	00000000
4	4	07001000
5	4	00000000
6	5	07100000
7	6	04200000
8	7	03400000
9	8	06000080
10	8	00000000
11	9	07004000
12	9	00000000
13	10	04100000
14	10	00000000
15	11	07020000
16	12	02200000
17	13	01400000
18	14	06008000
19	14	00000000
20	15	03040000
21	15	00000000
22	16	0B080000
23	16	00000000
24	16	00000000
25	17	0B200000
26	18	03400000
27	19	06010000
28	19	00000000
29	20	07100000
30	20	00000000
31	21	07008000
32	21	00000000
33	21	00000000
34	22	07200000
35	23	08400000
36	24	06004000
37	24	00000000
38	25	05100000
39	25	00000000
40	26	04080000
41	26	00000000
42	27	07010000
43	28	05200000
44	29	0A400000
45	30	06040000
46	30	00000000
47	31	02080000
48	32	00000040
49	33	07020000
50	33	00000000
51	33	00000000
52	34	03200000
53	35	04400000
54 (cycle 0 from next loop)	14 (mont)	06004000
55 (cycle 1 from next loop)	2	00000000
⋮	⋮	⋮

Bibliography

[1] Jean Claude Lapire. *Dependability: Basic Concepts and Terminology.* Springer, 1992.

[2] R. L. Rivest, A Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[3] ISO/IEC 9796. Information Technology-Security Techniques-Digital Signature Scheme Giving Message Recovery. International Organization for Standarization, 1991.

[4] R. L. Rivest, A. Shamir, and L.M Adleman. Cryptographic Communications System and Method. U.S. Patent #4,405,829, 1983.

[5] Victor S. Miller. *Use of Elliptic Curves in Cryptography.* Proceedings of Advances in Cryptology, Springer, pages 417-426, 1985.

[6] Neal Koblitz. Elliptic curve cryptosystems. In *Mathematics of Computation*, 48 (1987), American Mathematical Society pages 203-209, 1987.

[7] National Security Agency. The Case of Elliptic Curve Cryptography. https://www.nsa.gov/business/programs/elliptic_curve.shtml, last visited: April 2015.

[8] NIST Computer Security Division. Digital Signature Standard (DSS), FIPS 186-3. http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf, last visited: September 2014.

[9] Elliptic Curve Cryptography (ECC). https://www.certicom.com/, last visited: April 2015.

[10] IHP - Innovations for High Performance Microelectronics. http://www.ihp-microelectronics.com/en/start.html, last visited: April 2015.

[11] Debdeep Mukhopadhyay and Rajat Subhra Chakraborty. *Hardware Security: Design, Threats, and Safeguards.* Chapman & Hall/CRC, 2014.

[12] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2004.

[13] R.G.E. Pinch. Some primality testing algorithms. Notices of the American Mathematical Society, vol. 40, pages 1203-1210, 1993.

[14] Julio López and Ricardo Dahab. *Improved algorithms for elliptic curve arithmetic in GF(2^m).* Proceedings of the 5th Annual International Workshop on Selected Areas in Cryptography, Springer, pages 201-212, 1998.

[15] Paul Kocher, Joshua Jaffe, and Benjamin Jun. *Differential Power Analysis.* Proceedings of the 19th Annual International Cryptology Conference, Springer, pages 388-397, 1999.

[16] Jean-Sébastien Coron. *Resistance Agianst Differential Power Analysis For Elliptic Curve Cryptosystems.* Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pages 292-302, 1999.

[17] Peter L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. In *Mathematics of Computation*, 48 (1987), American Mathematical Society, 1987.

[18] Julio López and Ricardo Dahab. *Fast multiplication on elliptic curves over GF(2^m) without precomputation.* Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pages 316-327, 1999.

[19] Hikaru Sakamoto, Yang Li, Kazou Ohta, and Kazou Sakiyama. *Fault Sensitivity Analysis Against Elliptic Curve Cryptosystems.* Proceedings of the 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, IEEE Computer Society, pages 11-20, 2011.

[20] Huiyun Li, Keke Wu, Guoqing Xu, and Hai Yuan. *Simple Power Analysis Attacks Using Chosen Message against ECC Hardware Implementations.* Proceedings of the 2011 World Congress on Internet Security, IEEE, pages 68-72, 2011.

[21] Sahbuddin Abdul Kadir, Arif Sasongko, and Muhammad Zulkifli. Simple power analysis

attack against elliptic curve cryptography processor on fpga implementation. International Conference on Electrical Engineering and Informatics, 2011.

[22] Éric Brier and Marc Joye. *Weierstraß Elliptic Curves and Side-Channel Attacks.* Proceedings of the 5th International Workshop on Practice and Theory in Public Key Cryptosystems, Springer, pages 335-345, 2002.

[23] Christophe Clavier. Side channel analysis for reverse engineering (scare) - an improved attack against a secret a3/a8 gsm algorithm. IACR Cryptology ePrint Archive, 2004.

[24] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. *Templates vs. Stochastic Methods -A Performance Analysis for Side Channel Cryptanalysis.* Proceedings of the 8th International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pages 15-29, 2006.

[25] Eric Brier, Christophe Clavier, and Francis Olivier. *Correlation Power Analysis with a Leakage Model.* Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pages 16-29, 2004.

[26] Michael Hutter, Mario Kirschbaum, Thomas Plos, Jörn-Marc Schmidt, and Stefan Mangard. *Exploiting the Difference of Side-Channel Leakages.* Proceedings of the 3rd International Workshop on Constructive Side-Channel Analysis and Secure Design, Springer, pages 1-16, 2012.

[27] Pierre-Alain Fouque and Frederic Valette. *The Doubling Attack - Why Upwards is Better than Downwards.* Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pages 269-280, 2003.

[28] Naofumi Homma, Atsushi Miyamoto, Akashi Satoh, and Adi Shamir. *Collision-Based Power Analysis of Modular Exponentiation Using Chosen-Message Pairs.* Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pages 15-29, 2008.

[29] Mathieu Ciet and Marc Joye. *(Virtually) Free Randomization Techniques for Elliptic Curve Cryptography.* Proceedings of the 5th International Conference on Information and Communication Security, Springer, pages 348-359, 2003.

[30] Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. *A Practical Countermeasure against Address-Bit Differential Power Analysis.* Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pages 382-396, 2003.

[31] Elisabeth Oswald. *Enhancing Simple Power-Analysis Attacks on Elliptic Curve Cryptosystems.* Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pages 82-97, 2003.

[32] Ingrid Biehl, Bernd Meyer, and Volker Müller. *Differential Fault Attacks on Elliptic Curve Cryptosystems.* Proceedings of the 20th Annual International Cryptology Conference, Springer, pages 131-146, 2000.

[33] Zoya Dyka and Peter Langendörfer. *Area Efficient Hardware Implementation of Elliptic Curve Cryptography by Iteratively Applying Karatsuba's Method.* Proceedings of Design Automation and Test in Europe Conference, IEEE Society Press, pages 70-75, 2005.

[34] Steffen Peter. Evaluation of design alternatives for flexible elliptic curve hardware accelerators. Diplome Thesis-BTU Cottbus, 2006.

[35] IHP. Research - Project TAMPRES . http://www.ihp-microelectronics.com/de/forschung/drahtlose-systeme-und-anwendungen/abgeschlossene-projekte/tampres.html, last visited: October 2014.

[36] Synopsis - PrimeTime. http://www.synopsys.com/Tools/Implementation/SignOff/Pages/PrimeTime.aspx, last visited: September 2014.

[37] Marc Joye and Sung-Ming Yen. *The Montgomery Powering Ladder.* Proceedings of the 4th Internatinal Workshop on Cryptographic Hardware and Embedded Systems, Springer, pages 291-302, 2002.

[38] Zoya Dyka, Christian Wittke, and Peter Langendoerfer. *Clockwise Randomization of the Observable Behaviour of Crypto ASICs to Counter Side Channel Attacks.* To appear in: Proceedings of http://paginas.fe.up.pt/~dsd-seaa-2015/program/dsd-sessions-schedule/, 2015.