Elliptic Curve Cryptosystems on Reconfigurable Hardware

by
Martin Christopher Rosner

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical Engineering

May, 1998

Approved:

Prof. Christof Paar
ECE Department
Thesis Advisor

Prof. David Cyganski
ECE Department
Thesis Committee

Prof. Stanley Selkow
CS Department
Thesis Committee

Prof. John Orr
ECE Department Head

Abstract

Security issues will play an important role in the majority of communication and computer networks of the future. As the Internet becomes more and more accessible to the public, security measures will have to be strengthened. Elliptic curve cryptosystems allow for shorter operand lengths than other public-key schemes based on the discrete logarithm in finite fields and the integer factorization problem and are thus attractive for many applications.

This thesis describes an implementation of a crypto engine based on elliptic curves. The underlying algebraic structures are composite Galois fields $GF((2^n)^m)$ in a standard base representation. As a major new feature, the system is developed for a reconfigurable platform based on Field Programmable Gate Arrays (FPGAs). FPGAs combine the flexibility of software solutions with the security of traditional hardware implementations. In particular, it is possible to easily change all algorithm parameters such as curve coefficients, field order, or field representation.

The thesis deals with the design and implementation of elliptic curve point multiplication architectures. The architectures are described in VHDL and mapped to Xilinx FPGA devices. Architectures over Galois fields of different order and representation were implemented and compared. Area and timing measurements are provided for all architectures. It is shown that a full point multiplication on elliptic curves of real-world size can be implemented on commercially available FPGAs.

Preface

In this work I describe research that was conducted during my graduate studies at Worcester Polytechnic Institute. Inspired by the field of cryptography and reconfigurable computing, I hope that this thesis will be helpful in further development of research in this area.

The work presented in this book would not be possible without the help of many people. I would like to take this moment to thank everyone who contributed to this work. First, and foremost many thanks go out to my advisor *Prof. Christof Paar*. His advice and expertise resolved many hurdles that I encountered throughout this research. He was always able to point me in the right direction to find many answers. Consequently, I never lost confidence in my work. I would also like to thank GTE Government Systems who were able to partially sponsor my research at WPI.

I would also like to thank my colleagues who by now have become my good friends. Gregory Haskins showed me the ropes when I first joined the crypto research group at WPI. Jorge Guajardo was of great help when I was first putting together my first conference paper. His knowledge of the `gfops` libraries helped during the verification stage of initial designs. Many thanks go out to Jens-Peter Kaps who is currently finishing his MS work as well. Talking about our designs (even though they are completely different) and throwing ideas around has brought many sleepless nights. Also, many thanks go out to the system administrators at the ECE department, Murtaza Amiji and Brady Schulman, for their help with installation and upgrade of the Synopsys©applications on the HP workstations. Also, I thank my thesis committee who were kind enough to take the time to review this work. I'm certain their schedules are just as busy and I appreciate their comments and suggestions.

Last but not least, I would like to thank my parents and my friends for being patient with me and for their encouragement during my graduate studies. Special thanks go out to my future wife Joanna Zawadzki for her unconditional patience and understanding even as the time gets closer to our special date.

Thanks everyone!

—Martin

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Outline 3

2 Previous Work **5**

2.1 Hardware EC Implementations 5

2.2 Software EC Implementations 6

2.3 Overview on Finite Field Arithmetic Architectures 7

2.3.1 Finite Field Architectures for Cryptography 8

3 Arithmetic Operations **10**

3.1 Galois Fields 10

3.2 Addition 11

3.3 Parallel Subfield Multiplication 11

3.3.1 Binary Standard Base Multiplication 12

3.3.2 Composite Field Subfield Multiplication 12

3.4 Hybrid Multiplication 13

3.5 Squaring 15

4 Introduction to Elliptic Curves **17**

4.1 Historical Background 17

4.2 EC Crypto Engine Overview 18

4.3 EC Group Operation with Projective Coordinates 19

4.3.1 Definitions 19

4.3.2 Group Operation 20

4.4 Point Multiplication on Elliptic Curves 24

4.5 Elliptic Curve Cryptosystems 25

4.5.1 Diffie-Hellman Key Exchange 26

5 Methodology **28**

5.1 The Design Cycle 28

5.2 Tools 30

5.2.1 Xilinx Synopsys Interface 31

5.2.2 Simulation and Verification 32

5.2.3 Synthesis 33

5.2.4 Place and Route 33

6 General Design Considerations **35**

6.1 Synchronous vs. Asynchronous Design 35

6.2 Finite State Machines 37

6.3 Vendor Specific Design Components 38

6.4 Control Architecture 39

6.5 Processing Elements 39

6.6 Datapath Considerations 40

6.6.1 Routing Topologies 40

6.6.2 Xilinx FPGA 42

6.7 Serial vs. Parallel Computation Consideration 45

7 Design Overview **47**

7.1 Design Overview 49

7.2 Storage 50

7.3 Data/Address Select 52

7.4 Datapath Requirements 54

7.4.1 Combinatorial Components 55

7.4.2 Clocked Resources 55

7.4.3 System Timing Analysis 56

8 Arithmetic Level **59**

8.1 Multiplication in Hardware 59

8.2 Addition in Hardware 62

8.3 State Machine Encoding 64

9 Group Operation Level **65**

9.1 Double Sequence Operations 66

9.1.1 Precomputation Stage 67

9.1.2 Computation Stage 71

9.1.3 Summary 75

9.2 Add Sequence Operations 77

9.2.1 Precomputation Stage 78

9.2.2 Computation Stage 82

9.2.3 Summary 86

9.3 State Machine Encoding 88

10 Encryption Level **89**

10.1 Point Multiplication 89

10.2 State Machine Encoding 92

10.3 Point Multiplication Complexity 92

11 Results And Timing Analysis **94**

11.1 Introduction 94

11.2 Metric 95

11.3 Modules 97

11.3.1 Combinatorial Components 97

11.3.2 Clocked Components 102

11.4 System 106

12 Summary and Recommendations **109**

12.1 Summary 109

12.2 Recommendations for Future Research 110

12.2.1 New Design Considerations 110

12.2.2 Implementation Alternatives 111

12.3 Concluding Remarks 112

A Test Bench Sample **113**

B C Model **117**

C Synosys Script **132**

D Simulation Results **136**

D.1 $GF((2^4)^9)$ Multilication Simulation 136

D.2 $GF((2^4)^9)$ Double Sequance 138

D.3 $GF((2^4)^9)$ Add Sequance 145

References 159

List of Tables

4.1 Operation count for point doubling and point addition 23

6.1 Routing per CLB in XC4000 devices 45

7.1 Operation of Switch 1 52

7.2 Operation of Switch 2 53

7.3 Operation of Switch 3 54

9.1 Double_fsm precomputation sequence 69

9.2 Double_fsm computation sequence 72

9.3 Add_fsm precomputation sequence 80

9.4 Add_fsm state operations 84

10.1 IO_fsm state operations 90

10.2 Load state of IO_fsm 91

11.1 Addition results 98

11.2 Switch 1 results 99

11.3 Switch 2 results 100

11.4 Switch 3 results 101

11.5 Switch 5 results 101

11.6 RAM bank results 102

11.7 Multiplication results 103

11.8 I/O state machine results 104

11.9 Double state machine results 105

11.10Add state machine results 106

11.11Point multiplication after synthesis 107

11.12Point multiplication results 108

List of Figures

3.1 General structure of a hybrid multiplier in $GF((2^n)^m)$ 14

3.2 Serial multiplier optimized for consecutive square operations 15

4.1 System hierarchy of crypto engine 19

5.1 Design flow 32

6.1 Synchronous design example 37

6.2 XILINX programmable switch matrix 41

6.3 XILINX SRAM based FPGA structure 43

6.4 CLB structure (simplified) 44

6.5 Top level design view 46

7.1 System hierarchy 48

7.2 System diagram 49

7.3 Read/Write cycle for dual ported RAM 51

8.1 Memory access for multiplication 60

8.2 Arithmetic level FSM for Galois field multiplication 61

8.3 Memory access for addition 63

8.4 Arithmetic level FSM for GF addition 63

9.1 Internal architecture of control 66

9.2 Memory allocation table for double precomputation stage 69

9.3 Memory allocation for double computation stage 73

9.4 Memory usage for addition precomputation stage 81

9.5 Memory usage for addition computation stage 85

10.1 I/O finite state machine 91

D.1 $GF((2^4)^9)$ multiplication 137

D.2 $GF((2^4)^9)$ double sequance 139

D.3 $GF((2^4)^9)$ double sequance (cont.) 140

D.4 $GF((2^4)^9)$ double sequance (cont.) 141

D.5 $GF((2^4)^9)$ double sequance (cont.) 142

D.6 $GF((2^4)^9)$ double sequance(cont.) 143

D.7 $GF((2^4)^9)$ double sequance (cont.) 144

D.8 $GF((2^4)^9)$ add sequance 146

D.9 $GF((2^4)^9)$ add sequance (cont.) 147

D.10 $GF((2^4)^9)$ add sequance (cont.) 148

D.11 $GF((2^4)^9)$ add sequance (cont.) 149

D.12 $GF((2^4)^9)$ add sequance (cont.) 150

D.13 $GF((2^4)^9)$ add sequance (cont.) 151

D.14 $GF((2^4)^9)$ add sequance (cont.) 152

D.15 $GF((2^4)^9)$ add sequance (cont.) 153

Chapter 1

Introduction

1.1 Motivation

The latest breakthroughs in reconfigurable hardware technology are making reconfigurable computing more attractive to a wide range of applications. Today's programmable devices can accommodate very large digital designs with performances suitable in many high speed applications. At the same time, Internet popularity is growing very rapidly with applications ranging from computer/voice networks, electronic commerce and electronic banking. An open environment such as the Internet presents a threat to its users by compromising the privacy and integrity of every transaction. The necessity for security has fueled research in the area of cryptographic protocols and encryption algorithms. Since the Internet provides a diverse environment of heterogeneous systems, it is impossible to find one algorithm that meets the needs of all users. Consequently the need for a versatile approach to cryptographic services is obvious. Luckily, advancements in reconfigurable computing bring the possibility of reconfigurable cryptographic implementations into the real world.

Reconfigurable devices are of particular interest when considered for the use in cryptographic applications because of high degree of flexibility when compared to traditional ASIC solutions. Most modern cryptographic protocols such as SET and IPSEC are defined to be algorithmically independent. This means that several algorithms can be used for the same security service. For instance, a given protocol may allow RSA, DSA, or elliptic curves as digital signature algorithm. Moreover, today's cryptographic systems often rely on a hybrid approach that utilizes both private and public-key schemes. With configurable computing, it is possible to reuse a device to do both tasks faster than a software solution. A third reason why reconfigurable devices are attractive for cryptographic applications is that virtually all parameters of the design can be altered. For example, implementing elliptic curve (EC) systems in reconfigurable environment means that we have the capability to alter the curve parameters for each individual encryption stream. Also, the underlying arithmetic functions such as the finite field multipliers and adders can be changed with respect to field order or basis representation. Thus implementations based on reconfigurable hardware preserve the flexibility of software solutions while providing the security of a hardware solution. Also, with the introduction of partially reconfigurable devices, soon it will be possible to accommodate private and public-key schemes on the same device and be able to reconfigure only the desired functions while the device is functioning.

The work described here presents an elliptic curve (EC) implementation in Field Programmable Gate Arrays (FPGAs). At the time of this work there has not been any other documented effort in this particular area. Elliptic curves as cryptographic algorithms have been studied since the mid-1980s. The use of elliptic curves in cryptography is advantageous for many reasons. Elliptic curve cryptography allows for shorter key lengths without compromising the security of the system. In comparison to more conventional methods of public-key encryption such as RSA and systems based on the discrete logarithm problem which necessitate key lengths of about 1024 bits, EC systems use 160 bit operands. This translates to increased performance over both types of public-key algorithms. From a security standpoint, EC systems provide better long term security due to the lack of sub-exponential attacks which can be applied to the DL problem in finite fields. Finally, EC encryption is currently being strongly reviewed for standardization by the IEEE standards committee in P1363 [1].

1.2 Thesis Outline

We begin this thesis with a description of previous work related to this research. Thus, Chapter 2 summarizes hardware and software implementations of elliptic curve systems. In addition, recent research in finite fields arithmetic architectures is presented.

Chapter 3 dives right into theory with explanation of different arithmetic constructs considered for the realization of our elliptic curve system. This chapter is rather short but it provides a good background for Chapter 4 that describes elliptic curves for cryptographic protocols. In this chapter, some history and background is provided followed by explanation of elliptic curve group operation.

In Chapter 5, the entire design and implementation cycle is described. We felt strongly that the methodology chapter is quite necessary considering the very practical nature of this work. This chapter presents a road map for the entire research project. First, a brief overview is given describing how things were done and how one task relates to another. In addition, the tools that were used during the course of the project are explained. Associated with these tools are procedures for the development of each step of the design cycle.

In Chapter 6, reconfigurable hardware is described and few design considerations are discussed. More specifically, this chapter describes different design approaches that have been considered during the course of the project. Advantages and disadvantages of these options are presented in the context of FPGA architecture.

Chapters 7, 8, 9, 10 describe the final design. Chapter 7 presents a design overview and describes the individual components of the system architecture. Chapters 8, 9, and 10 provide a bottom-up approach in describing the control hierarchy of the system architecture. In Chapter 8 operations in $GF((2^n)^m)$ are described form a control point of view. Then, Chapter 9 develops the sequence that is necessary to realize group operation on EC. Finally, Chapter 10 presents the top level control structure that defines I/O operations and realizes point multiplication by a scalar integer.

Chapter 11 discuses our results and provides absolute timing analysis of the system. Thus timing results for point multiplication and individual double and add operations are derived from gathered data. Chapter 12 quickly concludes this written work with a short summary and recommendations for future research in this area.

Chapter 2

Previous Work

Previous work that aided in the development of this design include hardware and software realizations of point multiplication as well as implementation of Galois field arithmetic. The following summarizes previous work in these areas.

2.1 Hardware EC Implementations

Hardware realization of EC system results in higher performance and security at the expense of higher cost and reduced flexibility. Also, because of the large operands necessary in cryptographic applications, hardware solutions exhibit slower development cycles resulting in relatively few reports on EC in hardware. A hardware implementation of elliptic curve cryptosystem has been described as a co-processor unit by [6]. This VLSI implementation utilized optimal normal base representation for arithmetic in $GF(2^{155})$. The use of projective coordinates eliminated the need for inversion which is the most costly operation. Similar approaches to the hardware realization were described in [25] and [26]. Our approach differs from previously described implementations since we use composite field arithmetic in polynomial base representation.

2.2 Software EC Implementations

Software implementations of EC cryptosystem include [39, 11, 12, 45]. In [39], an elliptic curve system is implemented for a key exchange protocol. The implementation is simplified by choosing the curve parameter a equal to zero. The system architecture relies on arithmetic in $GF(2^{155})$ using polynomial representation and an optimized inversion algorithm based on Euclidean division. The implementation performed multiplication of a new elliptic curve point in 7.8 milliseconds on a DEC Alpha 3000 RISC machine.

Composite fields arithmetic has also been utilized in some previous elliptic curve implementations. Two different versions of this method were introduced. In [12], an elliptic curve cryptosystem was based on arithmetic in $GF((2^8)^{13})$ using a mixed normal/polynomial base representation. More recently, [45] and [11] describe elliptic curve cryptosystem implemented over $GF((2^{16})^{11})$. In both contributions different subfield arithmetic methods are analyzed and optimized for the specific implementation. Both contributions use a polynomial base representation and look-up tables for subfield arithmetic. In [45], inversion is achieved through a version of the Euclidean algorithm, whereas [11] uses a method based on exponentiation.

2.3 Overview on Finite Field Arithmetic Architectures

There are two main areas of application for finite field arithmetic: channel coding, in particular for the wide-spread Reed-Solomon codes [44], and public-key cryptography [41]. Although channel codes and cryptographic algorithms both make use of Galois fields, the field orders needed differ dramatically: channel codes are typically restricted to arithmetic with field elements which are represented by up to eight bits, whereas public-key algorithms rely on field sizes of several hundred bits. The majority of publications concentrate on finite field architectures for relatively small fields suitable for the implementation of channel codes.

Multiplication in $GF(2^k)$ is usually considered the crucial operation which determines the speed or throughput of a crypto system. Finite field architectures can be classified into bit serial (one output bit per clock cycle) and bit parallel ones (all output bits are computed within one clock cycle.) The majority of schemes are based on either of these two types. Architectures which are of hybrid-type (partially serial, and partially parallel), as used in this work, have been introduced in [36].

Bit parallel architectures tend to be faster than bit serial ones. According to the space-time trade-off paradigm, however, the former ones require more chip area in VLSI implementations. Bit serial multipliers have a space complexity of order $\mathcal{O}(k)$ for arithmetic in $GF(2^k)$. Bit parallel architectures usually have $\mathcal{O}(k^2)$ elementary gates as a lower complexity bound. More recently, however, new types of bit parallel architectures have been proposed with complexities below the k^2 bound [2, 3, 38, 33, 34]. These architectures are either based on multiple field extensions or on fast convolution methods such as the Karatsuba-Ofman algorithm (for an overview see [32, Chapter 3].)

Another classification of Galois field architectures is possible with respect to the base representation of field elements. The most popular bases are standard (or polynomial or canonical), normal base, and dual base. Each base representation has certain advantages; polynomial and dual base representations are well suited for bit parallel multipliers, whereas normal base representation allows for very efficient exponentiation.

There have been a few attempts to compare different types of arithmetic architectures for Galois fields. The focus is mainly on architectures for channel codes. In [16] multipliers for the field $GF(2^8)$ are compared for polynomial, dual, and normal representation. In [17] architectures are compared from a high-level description point of view. Again, the multipliers in the three different base representations are compared. A study in [35] compares architectures for the fields $GF(2^k)$, $k = 8, 16, 24, 32$, where architectures in polynomial, dual, normal basis, and with multiple field extensions are considered.

In [37], configurable computing platforms were used to compare various bit parallel Galois Field multipliers in FPGAs and EPLDs. The work done in this area has shown that bit parallel architectures are suitable for reconfigurable devices.

2.3.1 Finite Field Architectures for Cryptography

There is a relatively small number of published works on Galois field architectures which are especially designed for cryptographic applications. Many of the bit serial architectures mentioned in the previous section, however, also extend to cryptographic applications. The $\mathcal{O}(k^2)$ complexity bound of parallel multiplier architectures would result in unrealistically large arithmetic units for most public-key algorithms. So far, normal base and polynomial base representations have been used for cryptographic applications. Optimal normal base representations [30] are of special interest in this context because of their moderate complexity.

There are three relevant reported implementations which gain their security from the discrete logarithm (DL) in finite fields. Reference [9] deals with various aspects of bit serial architectures in Galois fields for cryptographic applications. An implementation of an exponentiation unit in $GF(2^{333})$ using polynomial base representation allows a data throughput of 15 kb/sec. Reference [4] contains a detailed description of an implementation of an exponentiation unit in the field $GF(2^{593})$. The implementation uses an optimal normal base representation of field elements. The reported maximum throughput is 300 kb/sec. In addition, there is the early description of an implementation of a cryptosystem over $GF(2^{127})$ [48]. This field order, however, does not provide adequate security against today's powerful DL attacks. The hybrid architecture used in our design was introduced for cryptographic applications in [36]. It will be reviewed in Section 3.4.

Chapter 3

Arithmetic Operations

Our EC implementation utilizes Galois Fields of characteristic 2 with a standard base representation. More specifically, we use a relatively new architecture type which is based on composite fields GF$((2^n)^m)$. Composite fields allow faster arithmetic architectures as described in [36]. Our implementation utilizes the multiplication module also to perform squaring in order to reduce the number of processing elements (PEs) and the routing to and from each PE.

3.1 Galois Fields

We assume that the reader is familiar with arithmetic in Galois fields (see e.g., [24]). In the following, we introduce the notation used throughout this thesis. Let $GF(2^n)$ denote the subfield with field polynomial $Q(y) = y^n + \sum_{i=0}^{n-1} q_i y^i$ where $q_i \in GF(2)$. Also, let $GF((2^n)^m)$ denote the composite field with the field polynomial $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$ where $p_i \in GF(2^n)$. In our implementation, we choose the subfield to be small ($n = 4, 8$) so that a parallel multiplication architecture in the subfield is possible with a small area utilization. We consider composite fields with two field extensions of degree n and m. Field elements are represented as polynomials with maximum degree $m-1$ over $GF(2^n)$.

3.2 Addition

Addition in $GF(2^n)$ is a very simple operation. Adding two polynomials where the coefficients are reduced modulo 2 is accomplished with a bitwise XOR function. Furthermore, adding polynomials based on composite fields does not complicate this operation as each subfield element also has to be XORed. Thus, addition in $GF((2^n)^m)$ requires $n \cdot m$ XOR gates and can be computed in one clock cycle in addition to one clock cycle for memory access.

3.3 Parallel Subfield Multiplication

In our EC implementation two types of multiplication schemes were applied for subfield multiplication. One is based on a binary standard base representation and the other utilizes composite fields. Applying two different techniques for parallel subfield multiplication can be useful when mapping these architectures to reconfigurable devices since different types of devices may yield better or worse performance for a given multiplier. The initial research that resulted from these tests was presented in [37]. Since the complexity of parallel multipliers grows exponentially with the width of the operands [23], a maximum of eight bits was used for the subfield extension n.

3.3.1 Binary Standard Base Multiplication

We used the multiplication architecture introduced by Mastrovito in [22, 23]. Multiplication of $A(y) \cdot B(y) = C(y) \bmod Q(y)$ is performed in $GF(2^8)$ by realizing polynomial multiplication and reduction in one step. All coefficients of the polynomials are reduced modulo 2 and the resulting polynomial $C'(y)$ is reduced modulo $Q(y)$. This can be done through the following matrix multiplication:

$$C = \mathbf{Z}B = \begin{pmatrix} f_{0,0} & \cdots & f_{0,n-1} \\ \vdots & \ddots & \vdots \\ f_{n-1,0} & \cdots & f_{n-2,n-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}. \tag{3.1}$$

where the matrix $\mathbf{Z}$ is named "product matrix"; $\mathbf{Z} = f(A(y), Q(y))$. We realized elliptic curve implementations with two different subfields, $GF(2^4)$ and $GF(2^8)$. The four bit version has $Q(y) = y^4 + y + 1$ as the irreducible polynomial and the eight bit one uses $Q(y) = y^8 + y^5 + y^3 + y^2 + 1$ for its irreducible polynomial. In both cases, the multipliers are implemented inside the hybrid multiplier architecture described in Section 3.4.

3.3.2 Composite Field Subfield Multiplication

In the last few years multiplication techniques over $GF((2^n)^m)$ have been developed using multiple field extensions [3, 33]. Composite fields are Galois fields with two extensions of degree n and m. Field elements are represented as polynomials with maximum degree $m - 1$ over $GF(2^n)$.

The eight bit composite field multiplier was considered such that $GF(2^8) \cong GF((2^4)^2)$. This means that the hybrid architecture for our EC implementation used a field

$GF(((2^4)^2)^m)$, where $GF((2^4)^2)$ was the composite subfield. We used the following approach for bit parallel multiplication in $GF((2^4)^2)$. The ground field $GF(2^4)$ has a primitive polynomial $Q(y) = y^4 + y + 1$. The second subfield extension has the primitive polynomial $R(z) = z^2 + z + \omega^{14}$ where ω is the primitive element in $GF(2^4)$ such that $Q(\omega) = 0$. Multiplication of two field elements $[a_0 + a_1 z][b_0 + b_1 z] \bmod R(z)$ can be realized as [33]:

$$\begin{aligned} C(z) &= A(z)B(z) \bmod R(z) \\ &= [a_0 b_0 + \omega^{14} a_1 b_1] + z[(a_0 + a_1)(b_0 + b_1) + a_0 b_0] \qquad (3.2) \\ &= [c_0 + c_1 z]. \end{aligned}$$

Thus, the composite subfield multiplier for $GF((2^4)^2)$ utilizes Galois field arithmetic in $GF(2^4)$. This is a different approach from the eight bit Mastrovito's multiplier since it computes the result for $GF(2^8)$ by using only $GF(2^4)$ arithmetic.

3.4 Hybrid Multiplication

The parallel multiplier architectures described in Sections 3.3.1 and 3.3.2 can only be implemented for relatively short operand lengths on FPGAs due to the high area requirements. Thus, to implement a multiplication operation in fields such as needed for elliptic curve system, e.g., with ≈ 160 bits, it is necessary to utilize a serial multiplier based on a linear feedback shift register. This section details such architecture.

The composite field multiplier used in the EC implementation was described in detail in [36]. This architecture is based on arithmetic in an extension field of $GF(2^n)$. The extension degree is denoted by m, so that the field can be denoted by $GF((2^n)^m)$.

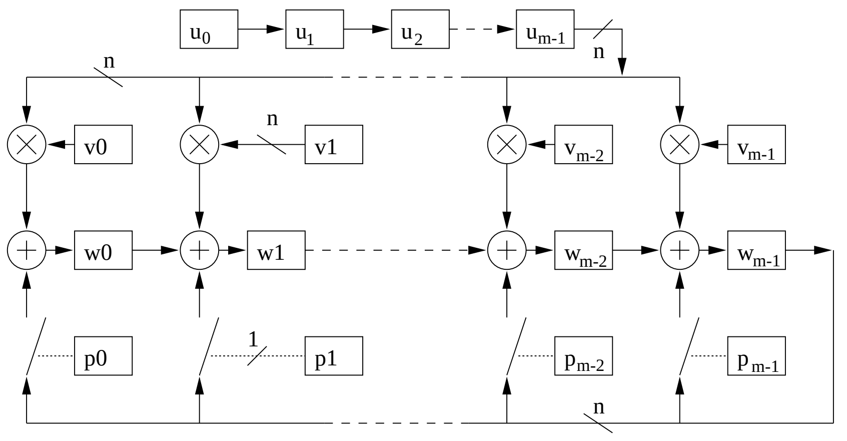

Figure 3.1: General structure of a hybrid multiplier in $GF((2^n)^m)$

For a standard basis multiplier, two field elements U, V are considered:

$$\begin{aligned} U(x) &= u_{m-1}x^{m-1} + \cdots + u_1 x + u_0, \\ V(x) &= v_{m-1}x^{m-1} + \cdots + v_1 x + v_0, \end{aligned}$$

where $u_i, v_i \in GF(2^n)$. Field multiplication with the two elements is performed by the operation $W(x) = U(x) \times V(x) \bmod P(x)$, with W being the product element. We restricted ourselves to irreducible extension field polynomial $P(x)$ with binary coefficients so that $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i, p_i \in GF(2)$. Note that such polynomial always exists if $\gcd(n, m) = 1$. A possible hardware realization for this operation, polynomial multiplication modulo the field polynomial, is shown in Figure 3.1. At the kernel of the architecture is a linear feedback shift register (LFSR) of width n and length m. The registers of the LFSR hold the w_i coefficients. The coefficients p_i of the field polynomial are the feedback coefficients of the the LFSR. The feedback coefficients are fixed in our implementation eliminating the need for the registers holding the feedback polynomial and the AND gates near these registers.

Since our implementation considers composite field architectures, all connections in Figure 3.1 are n bit wide buses and all arithmetic is performed in the subfield $GF(2^n)$.

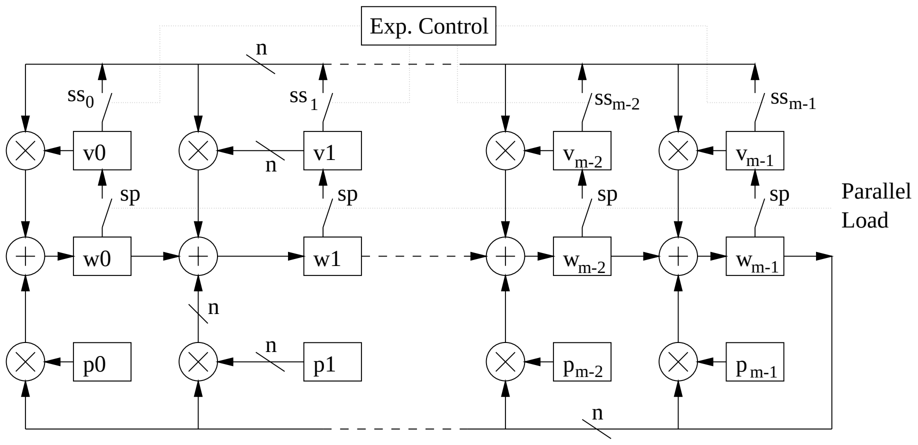

Figure 3.2: Serial multiplier optimized for consecutive square operations

Assuming bit parallel architectures for the subfield multiplication and addition in the LFSR, the result is computed in m clock cycles. Thus multiplying $m \cdot n$ bit operands requires m clock cycles in addition to an extra clock cycle for memory operation.

3.5 Squaring

The multiplier architecture described in Section 3.4 can be modified to improve performance when consecutive squaring operations are frequent. This can be accomplished by providing a path back from the result registers to the operand registers. This is shown in Figure 3.2. The path from $W(x)$ to $V(x)$ is controlled by a series of switches that are enabled only when consecutive squares are issued. Implementing this feature reduces memory access between each square operation to only one. Consequently, a clock cycle is saved for each consecutive square operation. Although this solution improves performance, our implementation of the EC engine does *not* realize such architecture due to limitations of routing resources in the FPGA. This alternative requires $n \cdot m$ additional paths allocated for the hybrid multiplier resulting in very long compilation times of the place and route tools. This improvement however becomes feasible with increased routing resources and better place and route tools or an ASIC implementation.

Chapter 4

Introduction to Elliptic Curves

In this chapter we will introduce Elliptic Curves (EC) and describe how they can be used to implement a public-key cryptosystem. Of particular interest are curves over fields of characteristic 2 when implementing them in a digital system. Therefore special emphasis is placed on the description of these curves. Finally, this chapter describes elliptic curves over projective coordinates since this contribution realizes such implementation.

4.1 Historical Background

Elliptic curves have been around for a long time in pure mathematics. In 1986 and 1987 elliptic curves have been proposed for cryptographic purposes [29, 19]. Two basic arguments make the use of elliptic curves quite attractive. First, with elliptic curves a wide variety of abelian groups could be formed allowing much more flexibility. In other words, there are many groups that can be used for a discrete logarithm one-way function. Second, there seems to be no sub-exponential attack known for solving the DL problem generated over elliptic curves. However the group operation is considerably more complex than the group operation of systems based on the DL in finite fields. This is the reason why EC have not initially caught on in the cryptographic community from an implementation point of view. Because of this increased complexity more research was necessary to show that EC implementation will generate a secure and feasible protocol.

From an implementation perspective, EC over $GF(2^n)$ can prove to be very practical. Implementing $GF(2^n)$ in a digital system is attractive due to the “binary” nature of the subfield $GF(2)$. Currently EC are being standardized by IEEE and ANSI after many years of research by the, sometimes sceptical, crypto community. EC provide for a shorter key and operation lengths making them attractive for many implementation. As computational power increases and attack algorithms improve very rapidly, it will be necessary to improve security by increasing the width of the operands. From an implementation point of view, it is more feasible to widen data paths for EC systems as they are much smaller than systems based on the DL in multiplicative groups in finite fields or RSA. In fact, the implementation described here is based on a slice architecture making it easier to increase the width of data in the EC engine.

4.2 EC Crypto Engine Overview

Throughout the rest of the thesis a general algorithm model is followed for our EC crypto engine. This model is shown in Figure 4.1. In Figure 4.1, the entire process is divided into three levels.

The encryption level defines the I/O interface as well as the algorithm for achieving point multiplication. The operation level defines control sequences necessary to re-

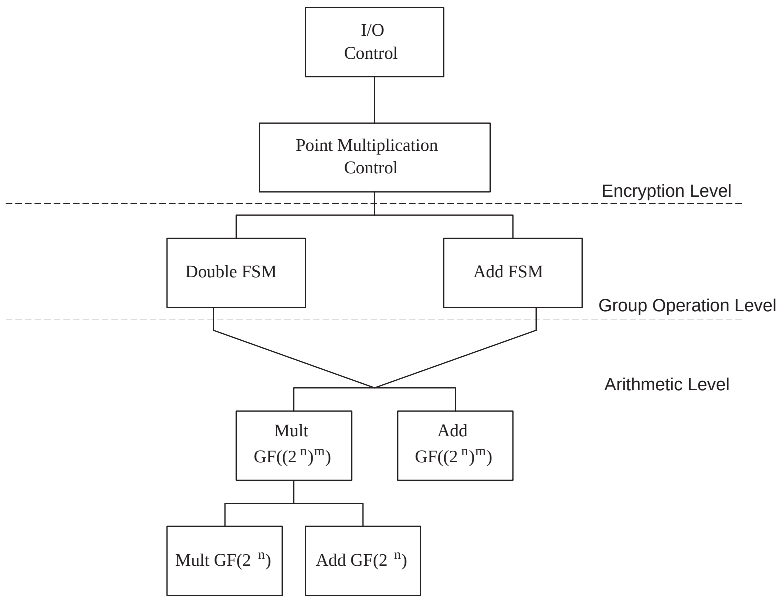

Figure 4.1: System hierarchy of crypto engine

alize point addition and point doubling. Finally, the arithmetic level describes the individual functions that are instanced by the double or add protocols. The theoretical background for the encryption and group operation level will be developed in the subsequent sections of this chapter. The theoretical background for the arithmetic level was presented in Chapter 3.

4.3 EC Group Operation with Projective Coordinates

4.3.1 Definitions

Some definitions may be helpful when describing the EC constructs. A few are presented below.

- Group — a group $(G, +)$ consists of the set G with an operation "$+$" on this group satisfying the following rules:

  1. group operation is associative and closed
  2. there is an identity element γ such that $a + \gamma = a$ for all $a \in G$
  3. there is an inverse element for all $a \in G$ such that $a + a^{-1} = \gamma$

- One way function — a function that provides for a computationally easy mapping from set X to set Y for all $x \in X$ but becomes computationally infeasible when mapping an element from set Y to set X for most $y \in Y$.

- Discrete logarithm (DL) problem — a particular one-way function with $x, y \in G$ such that the discrete logarithm of x to base y, denoted by $\log_y(x)$, has a unique integer solution z where $x = y^z$.

4.3.2 Group Operation

The standard formulae for adding two points on an elliptic curve with affine coordinates require 1 inversion which can be very costly in fields of order $\approx 2^{160}$ [28]. As a result, other solutions have been developed [28, 40] that eliminate the need to invert in such large fields. This implementation realizes elliptic curves with projective coordinates. In the following, projective coordinate equations are derived for EC equations over fields of characteristic 2.

A non-supersingular curve over Galois fields with characteristic two is defined as:

$$y^2 + xy = x^3 + a_2x^2 + a_6 \tag{4.1}$$

It is important to mention here that non-supersingular curves are of particular interest because they are not susceptible to sub-exponential attacks. Equation (4.1) together with the point at infinity $\mathcal{O}$ forms an elliptic curve where $a_2, a_6 \in GF(2^k)$, $a_6 \neq 0$ [28]. Points (x, y) which fulfill (4.1) together with the operation "+" generate a group that can be used to implement a public-key scheme such as the one described in Section 4.5. An extensive description of elliptic curves can be found in [40]. If points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ are added, such that $R = (x_3, y_3) = P + Q$, two cases must be distinguished. In the first case P and Q are different points (point addition). In the second case P and Q are identical, i.e., $x_1 = x_2$ and $y_1 = y_2$, (point doubling). Thus we have two instances of the + group operation defined as follows [28]:

$$x_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a_2, & \text{if } P \neq Q \\ x_1^2 + \frac{a_6}{x_1^2}, & \text{if } P = Q \end{cases}$$
$$\text{and} \tag{4.2}$$
$$y_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})(x_1 + x_3) + x_3 + y_1, & \text{if } P \neq Q \\ x_1^2 + (x_1 + \frac{y_1}{x_1})x_3 + x_3, & \text{if } P = Q \end{cases}$$

From the above equations, it is apparent that inversion is necessary to solve for the third point on the curve. Since inversion is particularly costly in hardware, we will consider an alternate point representation. This can be done if the elliptic curves are considered with projective coordinates [28]. This transformation can be achieved by mapping the set of points onto a homogeneous cubic equation of the form [40]:

$$\text{E: } y^2z + xyz = x^3 + a_2x^2z + a_6z^3 \tag{4.3}$$

So points (x_1, y_1) and (x_2, y_2) are now represented by the point $P = (x_1 : y_1 : 1) \in E$ and point $Q = (x_2 : y_2 : 1) \in E$. Note that any two points are equivalent if $(x_1, y_1, z_1) = \lambda(x_2, y_2, z_2)$. Thus, dividing the coordinates of a point (x_1, y_1, z_1) by z_1 results in $(x_1/z_1 : y_1/z_1 : 1) \in E$ which is the inverse mapping from projective coordinates to affine coordinates. At this point, we can use the addition formulas in

Equation (4.2) to derive the equations for projective coordinates:

$$\begin{aligned} x_3' &= \frac{B^2}{A^2} + \frac{B}{A} + \frac{A}{z_1} + a_2, \\ y_3' &= \frac{B}{A}(\frac{x_1}{z_1} + x_3') + x_3' + \frac{y_1}{z_1}, \end{aligned} \tag{4.4}$$

where $A = (x_2z_1 + x_1)$ and $B = (y_2z_1 + y_1)$. Assigning $z_3 = A^3z_1$ and multiplying through the x_3 and y_3 coordinates will cancel the denominator portions from Equation (4.4), effectively removing inversion from the curve equations. This results in the following addition formulae:

$$\begin{aligned} x_3 &= AD, \\ y_3 &= CD + A^2(Bx_1 + Ay_1), \\ z_3 &= A^3z_1, \end{aligned} \tag{4.5}$$

where $C = A + B$ and $D = A^2(A + a_2z_1) + z_1BC$. These addition formulae can be implemented in 14 multiplications. With affine coordinates this operation would require 3 multiplications and one inversion. As a consequence, using projective coordinates is more efficient until a polynomial inversion can be accomplished with less than 11 multiplications.

The equations for point doubling ($P = Q$) can be derived in a similar manner resulting in:

$$\begin{aligned} x_3 &= AB, \\ y_3 &= x_1^4A + B(x_1^4 + y_1z_1 + A), \\ z_3 &= A^3, \end{aligned} \tag{4.6}$$

where $A = x_1z_1$ and $B = a_6z_1^4 + x_1^4$. Here, we sacrifice 5 multiplications for one inversion because we avoid two inversion computations by gaining 10 extra multiplications

Equations (4.5) and (4.6) define the EC group operation over projective coordinates. These equations are used in our EC implementation on reconfigurable hardware. They are thus crucial for the remainder of the thesis.

At this point it is possible to provide an exact count for the number of operations necessary to realize point doubling and point addition. In the case of point doubling (Equation (4.6)), the computation of the intermediate values requires 6 multiplications and 1 addition in $GF((2^n)^m)$. More specifically, computing A results in 1 multiplication, and B results in 5 multiplications and 1 addition. Once the intermediate values are calculated, x_3 is obtained with 1 multiplication, y_3 requires 3 multiplications and 3 additions, and z_3 is obtained with 2 multiplications. Thus, point doubling requires a total of 12 multiplications and 4 additions. Similar analysis of point addition can be performed to obtain an operation count for this sequence. Table 4.1 summarizes these complexity results for point doubling and point addition. One sees

Sequence	*Additions in* $GF((2^n)^m)$	*Multiplications in* $GF((2^n)^m)$
Point Double	4	12
Point Add	7	14

Table 4.1: Operation count for point doubling and point addition

that saving the inversion required for affine coordinates comes at the cost of more multiplications. Also, since the number of multiplications has increased, the designer is forced to use more temporary registers for the intermediate values as more data dependencies are present.

4.4 Point Multiplication on Elliptic Curves

The core operation in EC cryptosystems is point multiplication $l \cdot P$ where l is an integer and P is a point on the curve. A single point multiplication requires multiple computations of point addition ($P \neq Q$) and point ($P = Q$) doubling which were described in Section 4.3. The standard method for point multiplication is the double-and-add algorithm. For example, $\beta = 10 \cdot \alpha$ is computed as $\beta = 2(2(2\alpha) + \alpha)$ which requires three doubling and one addition operation. Calculating $l \cdot P$, where P is a point on the curve, will yield a new point on the curve. This procedure forms the basis for public key cryptography using EC. The double and add algorithm is analogous to the square and multiply algorithm used for exponentiation [18]. The algorithm is defined as follows:

Double and add algorithm:

```
1. $l$ is an integer such that $l = (l_r, l_{r-1}, \ldots, l_1, l_0)$ is the binary representation of $l$
   with most significant bit $l_r = 1$.
2. Copy original point to temporary variable: $temp \longleftarrow P$.
3. For $index$ from (r-1) downto 0 do:
    (a) DOUBLE: $temp \longleftarrow temp + temp$.
    (b) if $l_{index} = 1$, then also ADD: $temp \longleftarrow temp + P$.
4. return $temp$ which contains $l \cdot P$.
```

Chapter 10 describes the hardware implementation of this algorithm. In general, l has the same number of bits as the order of the point group used which in turn is approximately equal to the order of the underlying finite field due to Hasse's theorem

[40]. Thus, for an EC over the composite field $GF((2^n)^m)$, the multiplier l has $(n \cdot m)$ bits of length. Average case analysis of the double-and-add algorithm will yield the following number of double and add operations: one point multiplication requires $(n \cdot m) - 1$ double operations and $((n \cdot m) - 1)/2$ add operations [27]. Analysis of the double equations in projective space developed in Section 4.3 shows that 12 multiplications/squarings and 4 field additions are needed for one double operation. Similarly, an addition operation can be accomplished in 14 multiplications/squarings and 7 field additions. Consequently, one point multiplication requires $19 \cdot [(n \cdot m) - 1]$ field multiplications and $7.5 \cdot [(n \cdot m) - 1]$ field additions. Furthermore, since each multiplication/squaring requires $m + 1$ clock cycles and each field addition can be done in 2 clock cycles (please refer to Chapter 3), the total number of clock cycles necessary to compute one point multiplication is

$$\#\text{clkcyc} = 19[nm^2 + nm - m - 1] + 15[(nm) - 1]. \tag{4.7}$$

The results presented in this thesis provide the minimum clock period for our implementations. Using Equation (4.7) together with our results will provide the absolute timing required for one point multiplication for all implemented architectures.

4.5 Elliptic Curve Cryptosystems

As a brief example of how EC system can be used in public-key cryptosystem, this section will outline one of the most popular protocols. The Diffie-Hellman (D-H) key exchange protocol can be based on elliptic curves. It is important to realize that the protocol described below is only one example of a EC public-key protocol. In particular, digital signature and encryption protocols are also possible as outlined in the IEEE P1363 draft standard. All EC protocols have point multiplication as the central algorithmic component so that our crypto engine could be used for all EC protocols.

4.5.1 Diffie-Hellman Key Exchange

A more detailed description of the general D-H key exchange is presented in [27, 42]. The goal of this protocol is to establish a secret session key between to parties over an unsecure channel. The two parties, Alice and Bob, want to establish a secret key without Oscar (the adversary) being able to compute this key. During the setup stage Alice and Bob obtain the public parameter α with the coordinates $(x_\alpha : y_\alpha : z_\alpha) \in E$ which is a point on the elliptic curve. The rest of the algorithm proceeds as follows:

1a) Alice generates a random key:		1b) Bob generates a random key:
a_A (private)		a_B (private)
2a) Alice computes a new point:		2b) Bob computes a new point:
$\beta_A = a_A \cdot \alpha$ (public)		$\beta_B = a_B \cdot \alpha$ (public)
3a) Alice sends β_A to Bob	$\xrightarrow{\beta_A}$	
	$\xleftarrow{\beta_B}$	3b) Bob send β_B to Alice
4a) Alice computes:		4b) Bob computes:
$a_A \cdot \beta_B = a_A(a_B \cdot \alpha) = (x_a : y_a : z_a)$		$a_B \cdot \beta_A = a_B(a_A \cdot \alpha) = (x_b : y_b : z_b)$

After the final stage of the algorithm, Alice and Bob can compute the shared session key K_s as $K_s = x_a/z_a = x_b/z_b$. Oscar cannot regenerate the session key from the public parameters α, β_A, and β_B because the two random integers, a_A and a_B, generated by Alice and Bob are private and were never transmitted over the unsecure channel. The security of this scheme relies on the discrete logarithm problem for EC which is believed to be intractable if "secure" curves (non-supersingular, suited group order) over sufficiently large fields are chosen [27]. Once the session key is established between Alice and Bob, both parties can communicate securely using private key algorithm such as DES which generally allows very high encryption speeds.

Chapter 5

Methodology

This chapter describes the process through which the design was conceived, defined, implemented, and verified. The choice of tools and supporting devices is explained in this chapter. Also, some remarks on the performance and effectiveness of the tools are given.

5.1 The Design Cycle

The general design cycle for this work consisted of the following steps:

1. Research of arithmetic functions.

2. Research of elliptic curve constructs.

3. VHDL implementation of arithmetic functions.

4. Commitment to a specific implementation of elliptic curve field representation.

5. Design of point multiplication elliptic curve engine.
6. Logic verification of the design.
7. Synthesis and logic optimization.
8. Device specific realization (place and route).
9. Back-annotated verification of the design.

The order of steps outlined above is more or less accurate. At some point of the project, steps had to be retraced to ensure optimal or correct implementation. Since not all algorithms can be easily implemented in hardware, careful consideration of the implementation was necessary before committing to a specific option. By doing the initial research into Galois Field arithmetic operations and their implementations in hardware, a few guidelines were developed that aided in the choice of Galois field representation and elliptic curve point representation. More specifically, standard base representation for Galois field arithmetic was chosen and composite architectures were mapped to reconfigurable devices [37]. Furthermore, projective coordinate representation was chosen for the elliptic curve point representation. By using projective coordinates some inversion, which is by far the most complex operation, was avoided. Thus at the and of initial research, commitment was made to realize the elliptic curve engine with projective coordinates and standard base representation.

The next stage was the actual design of the digital system that realized the elliptic curve group operation. During this stage many revisions were made to better fit the design to a specific device (please refer to Chapter 6). The XILINX FPGA XC4000 family of devices was chosen as the target platform.

Because of the vast array of reconfigurable devices available today, VHDL implementations tend to become vendor specific. This is due to the fact that many vendors provide soft macros for predefined components. These macros take advantage of specific features in a given device making it nearly senseless to build equivalent functions with VHDL. By choosing the XC4000XL family of devices, it was now possible to identify useful macros for the elliptic curve design. Having the knowledge of these macros and the initial research into elliptic curve constructs, a digital system was developed in VHDL with macro instantiations.

Verification of the design was first performed on the logic level basis. This step assured the correct functionality if all combinatorial and net delays were ignored. Once the design was verified logically, synthesis and optimization was performed. Timing constraints were set for each component and different iterations were done until constraints were met. The next step was to actually map, place and route the design into reconfigurable device. The choice of a specific device within the XC4000 family depends on the area utilization report obtained through synthesis. Finally, the output of the place and route step was used to perform back-annotated simulation. This step verified the correct operation with net and combinatorial delays that resulted from the place and route process.

5.2 Tools

The entire design, with the exception of vendor specific soft macros, was entered in VHDL format. Once the design was developed in VHDL, boolean logic and major timing errors were verified by simulating the gate level description with Synopsys VHDL analyzer (`vhdlan`) version 1997.08. The next step involved synthesis of the VHDL code with Synopsys (`fpga_analyzer`) version 1997.08. The output of this step was an optimized netlist describing the gate level design in XILINX format. The most time consuming step was the compilation of the synthesized design with the place and route tools available from XILINX. The design presented routing challenges for the tools resulting in a few iterations with different clock constraints. The slow compile times associated with this step resulted in a limited amount of data that could be gathered during this process. This step was accomplished with the XILINX Design Manager tools version M1.3.7. The final step of the design flow was to verify the design once again but this time with the physical net, CLB, and pad delays introduced when the design was placed into a specific device. This final stage of the design was accomplished with the same test benches and simulation models that were used during the logic verification stage. Synopsys (`vhdlan`) was used once again to verify back-annotated designs.

5.2.1 Xilinx Synopsys Interface

Figure 5.1 presents a flow chart diagram of the design flow with Xilinx-Synopsys-Interface (XSI) tools. The XSI tools provide for a transitions between results obtained from Synopsys synthesis and the Xilinx place and route tools. The XSI module includes all libraries necessary for Synopsys `fpga_analyzer` to interpret gates into logical blocks so that synthesis can be performed at this level. The design_ware libraries provided by Xilinx are automatically instantiated when possible. For example, if the VHDL code contains addition "+" operations (such as the one used to increment counters), the Synopsys tool will utilize Xilinx design_ware libraries to instantiate soft macros for such adder units. Synthesis results include report files on area and timing utilization, design netlist and constraints that are used in the place and route process, and Synopsys design files that describe the entire system.

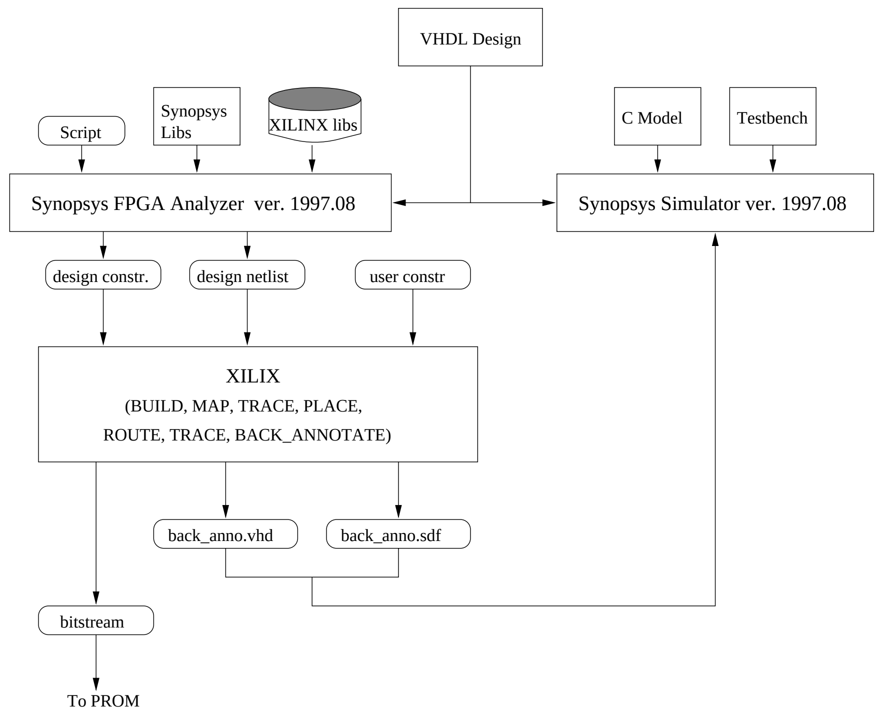

Figure 5.1: Design flow

5.2.2 Simulation and Verification

As previously stated, verification of the design is done at two points. First, it is applied to the initial VHDL design. This verifies only the logic without delays. The input to this verification process is a test bench written in VHDL, a model of the design written in C, and the actual VHDL design. The test bench is used together with the VHDL design to simulate the design. Then the results from the simulation are compared against results obtained from the C model. Logical verification is complete when all the test vectors are verified.

The post place and route verification uses the same C model and test bench (with few modifications). The VHDL input model to this stage is different. Here the VHDL model is obtained from the XILINX place and route tools. This VHDL model looks nothing like the original one and it includes a separate file defining all net, CLB, and port delays associated with the placed design. Once again, verification process involves testing all vectors against the C model results. A sample test bench can be found in Appendix A and the C model is included in Appendix B.

5.2.3 Synthesis

Synopsys synthesis tools have only been available to us in the past year. The documentation that accompanied these tools was quite extensive and very helpful. This and other literature helped in developing script files that could be launched from within the `fpga_analyzer`. These scripts would elaborate, compile, optimize the design, and prepare report summaries. A sample script file is provided in Appendix C. One advantage of running this tool on an HP station was that multiple jobs could be run concurrently resulting in faster turnaround and more time to try different optimization options.

5.2.4 Place and Route

The place and route tools from Xilinx were used on the HP workstation as well. The compilation proved to be very slow due to the complex routing task. Consequently, this step was moved to a Windows environment so fast Pentium-based PCs could be used. This is not to say that this tool runs better in Windows. It is just that the availability of better hardware forced the migration to the Windows environment. Even running on a 200MHz Pentium processor and 256MB of RAM, required several days (close to a full week) for the $GF((2^4)^9)$ design to be placed and routed.

The input to the place and route tools is a design netlist and constraints files generated by Synopsys, as well as possible user constraints file. The user constraints have higher priority over the Synopsys constraints and may include additional constraints relaxing the clock period or implementing pin assignment. As stated before, the output of this process is a bit-stream file that can be used to directly program the device and the back-annotated design that can be simulated for timing verification (please refer to Figure 5.1).

Chapter 6

General Design Considerations

In this chapter we will describe the general constructs that define a digital system and explore different topologies that may be suitable for FPGAs. Furthermore, the control, data, and processing units will be introduced as the basic building blocks of the (EC) implementation.

Before we describe the digital design of the entire EC cryptosystem, it is essential to outline some of the more pronounced decisions that have to be made when considering FPGA implementation. In this section we give a general overview of certain aspects of the design that have to be considered when mapping a digital system into a Look-Up Table (LUT) architecture such as the XILINX XC4000 devices.

6.1 Synchronous vs. Asynchronous Design

Although synchronous designs are more popular and easier to implement, asynchronous designs exhibit properties that are desirable in many digital systems. One major attribute of asynchronous designs is that handshaking between processing elements is more defined absolving the need for a centralized clock signal. Conversely, synchronous designs rely on a clock signal whose period must be greater than the longest combinatorial delay. Some other more prominent advantages of asynchronous designs are presented in [15]. However, asynchronous design have many disadvantages when considered in the context of FPGA design. As much as we would like to attempt a design of an (EC) digital system using asynchronous methodology, [13] suggests that such an approach necessitates elimination of all hazards, synchronization of events and very precise timing requirements of all functional blocks in the design. Timing glitches due to poor combinatorial design can be avoided with careful consideration. However, timing glitches produced by the place and route tools are, for the most part, very hard to control. It is these glitches that render asynchronous designs on FPGAs infeasible at this point. Few attempts have been made to develop reconfigurable devices capable of asynchronous operation [13, 14]. Thus, until such devices become readily accessible, FPGA implementations are limited to synchronous methodology.

Our implementation is developed in synchronous methodology. With this approach, the design becomes easier since the clock period is determined by the slowest combinatorial delay between two registers. Consequently timing glitches occurring during any transitions are allowed to settle before the computed values are registered. Figure 6.1 shows the general structure necessary to achieve synchronous operation. The implementation is mapped to an FPGA from Xilinx. More specifically, the XC4000 family devices were chosen according to the size and/or routing resources necessary for a given version of the implementation. The XC4000 FPGA is exceptionally suitable for synchronous architecture due to the abundance of flip-flops in the device. Each Configurable Logic Block (CLB) contains two flip-flops that can be configured as RAM or registers [47]. Since the cryptographic algorithm requires wide data paths and plenty

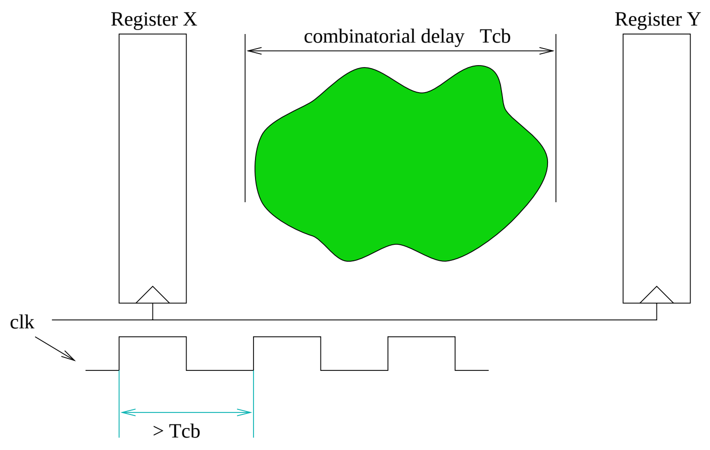

Figure 6.1: Synchronous design example

of storage capability, a register rich environment is very desirable.

6.2 Finite State Machines

The means for implementing state machines in the XC4000 family devices is another consideration that needs some elaboration at this point. In particular the state machine encoding style has to be chosen to fully utilize the specific FPGA resources. In a conventional ASIC design, binary or gray code state machine encoding is preferred for efficient and minimal design [43]. Conversely, "one-hot encoded" state machines implemented in ASIC device exhibit relatively large area with minimal gain in performance. This is not the case with FPGA devices. As described in [10], one-hot encoding becomes feasible when mapped to FPGAs due to the large number of flip-flops available in the device. For many designs, the CLBs are used as LUTs leaving the flip-flops untouched. Consequently, these resources can be utilized when one hot encoded state machine is implemented [10]. The advantage of a one-hot encoding lies in the representation of each state with an individual bit (flip-flop) resulting in decreased logic complexity associated with each state [7]. However, any FPGA device has a limited number of routing and area resources thus implementing a one-hot encoded state machine is advantageous only in cases where there are enough resources for the state machine and the rest of the design. Once the state machine becomes too complex, [7] suggests that an external RAM may be required to store the control program leaving only the sequencing mechanism inside the FPGA. This may become significant in the design of EC state machine since the number of operations necessary for point addition or point doubling is quite significant. The goal of our work was to implement entire system (including all storage elements) in the XILINX FPGA since this device provides for a very register rich environment.

6.3 Vendor Specific Design Components

Many FPGA manufacturers provide an array of arithmetic and storage macros that take advantage of specific features in a particular device. Xilinx provides a design-ware library filled with such macros in addition to LogiBlox components that can be tailored to the designer's specific need. Arithmetic functions such as adders use the fast carry chains available on the device. Also RAM/ROM elements mapped with available macros fully utilize the CLB structure to achieve the most performance out of a single CLB. In [10], many designs are analyzed to show the effectiveness of the available macro functions. Our implementation takes advantage of the LogiBlox functions for storage (RAM and registers) elements. Also, the design-ware library is invoked by Synopsys tools to implement arithmetic functions necessary for counters present in the control module.

6.4 Control Architecture

The larger number of data dependencies necessitates extra control logic to schedule the operations at the correct time. The control mechanism is composed of two state machines (Double_fsm and Add_fsm) that assert certain data segments to either feed the processing element (PE) or route the results from the processing element back to the RAM bank. The two state machines work independently and never concurrently since the double and add operations cannot be invoked at the same time. The Double_fsm and Add_fsm start processing when a signal is asserted by the IO_fsm that controls the double-and-add sequence. Depending on how many registers will be necessary to realize point addition and doubling, the control unit will also have to utilize large multiplexers that route one out of q register outputs to the PEs. The same type of scheme is needed on the other side of the PE to feed the correct register with the result of a certain operation. These large multiplexers can be avoided if the registers can be replaced with a RAM module. Our EC implementation uses LogiBlox RAM to minimize on the utilization of registers and decrease the required routing resources. The details of the control unit (CU) will be described in subsequent chapters.

6.5 Processing Elements

Since inversion is not necessary if projective coordinates are being used, the most costly operation becomes multiplication. Besides multiplication, addition is also required, however this operation is trivial as described in the previous chapter. Chapter 3 developed architectures for Galois field multiplication and addition. Furthermore, squaring optimizations were reviewed. The simplest approach for realizing point addition and doubling is to use the smallest number of PEs. Consequently, for the implementation of our design, only one multiplication PE and one addition PE is used. This approach also means that squaring is simply implemented with the multiplication PE. Finally, the implementation does not assume parallel processing of the multiplier PE and adder PE. The main reason for this is to reduce the control logic and minimize routing. Once a working model of the first implementation is accomplished more information regarding data routability and size of the final design will be available making it easier to make decisions about further implementations.

6.6 Datapath Considerations

Routing capabilities of different reconfigurable devices is an important issue that needs to be researched in more detail. Point addition and doubling requires multiplication of polynomials of degree ≈ 160. This means that the device has to be capable of routing and multiplexing 160 bits of data simultaneously. Consequently FPGA devices with different topologies will exhibit different delays for such design.

6.6.1 Routing Topologies

The EC implementation or any digital system for that matter, requires many different routing resources. Furthermore utilization of routing resources strongly depends on the placement of logic blocks. Thus running place and route tools with different timing, pad, or placement constraints will all result in different placement. This, in turn, will effect the use of routing resources. For example, taking the serial multiplier structure described in Section 3.4, and mapping it to a particular device will utilize long lines for the feedback path and short lines to route data between registers which are closer together. In addition, if the serial multiplier is too wide to fit in one column or one row, the placement tools will have to make a decision, based on timing or pad constraints, on where to break up the slice architecture so as to meet these constraints. It is very likely that the multiplier will be broken near the place where the first feedback path is taken. In any case, it is apparent that different routing resources are used for different purposes.

The XILINX FPGA devices have an array of different routing lines of which the shortest lines are most abundant. However the use of an interconnect switch matrix makes it possible to join two shorter lines in order to achieve longer connectivity. Figure 6.2 shows the programmable switch that is provided at the corner of each logic block in the XILINX FPGA. These structures provide great flexibility for the

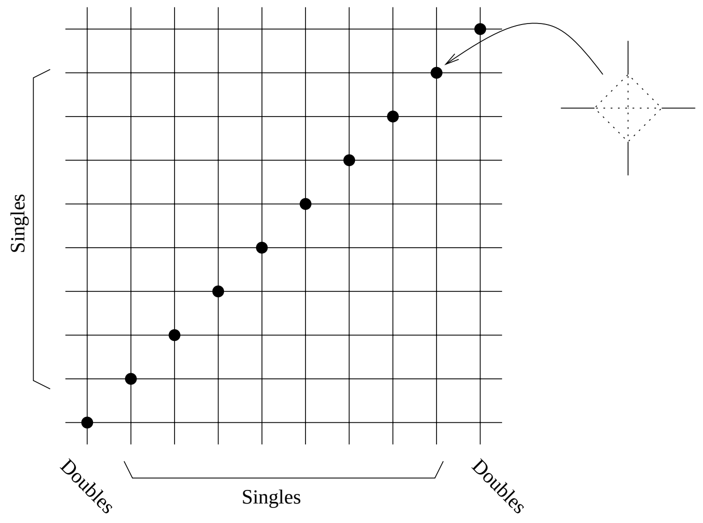

Figure 6.2: XILINX programmable switch matrix

routing tools and are the main reason why the XILINX devices can achieve very high utilization. The increased flexibility directly translates into longer compile times as the place and route tools have a much harder time converging to the optimal result.

6.6.2 Xilinx FPGA

The digital system developed in this work was mapped to a XILINX device. More specifically, the XC40ddXL devices were considered ("dd" specifies a particular device of given size). The XL devices are similar to the EX devices which are also available from XILINX. The only difference is that the XL devices have more CLBs (area) which in turn translates to more routing resources. For a detailed description of these and other devices from XILINX, the reader is encouraged to look in [47] as this section provides only a brief look at the structure of the XILINX XC4000 family devices.

The basic structure of the XC4000 family devices is shown in Figure 6.3. This structure is based on a fine grain approach which means that logic or storage functions are mapped into small blocks. Typically, these blocks are quite abundant in a single device with each block having very few inputs and even fewer outputs. With this fine grain approach, routing is a central issue as sufficient resources are necessary to provide connectivity between all CLBs. The XILINX FPGA incorporates a matrix of switch boxes that is placed over the CLB array. By programming the switch boxes during the configuration stage, it is possible to connect any two CLBs together. As a result of the fine grain architecture and the versatility of switch boxes, routing a particular design (especially a large design) becomes very difficult. These are a few of the reasons why long compilation times were experienced when trying to map the final design into the device. The advantage of a fine grain structure is that it provides for greater flexibility for the synthesis tools. Logic optimization becomes fine grained as well which typically results in better performance.

The functional unit inside the XILINX FPGA is a configurable logic block (CLB) which is shown in Figure 6.4. This block is composed of two 4-input (F and G blocks) and one 3-input (H block) function generators. These elements are simple

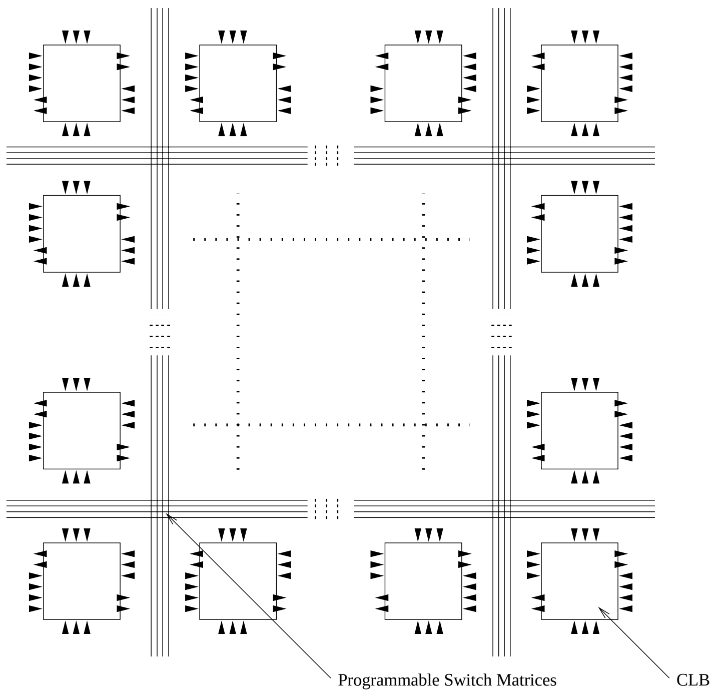

Figure 6.3: XILINX SRAM based FPGA structure

look-up tables (LUTs) that can perform four or three input functions. Thus, logic is not implemented in gates. This is an important point as one generator function can behave like a number of gates while exhibiting a delay of only one level. The CLB also has two flip-flops that can be implemented as RAM (as used in our implementation for storage) or registers (as used in our implementation inside the serial multiplier). With the LogiBlox application from XILINX, it is possible to implement a vast array of storage elements and other functions.

The main reason for choosing the XL (EX) family of devices is the increased amount of routing resources over the E devices. Routing in the XILINX FPGA is accomplished through a hierarchal structure. Thus each row or column of routing lines between CLBs has a number of different types of lines. These include single, double, quad, long, and global lines. Single lines route signals between adjacent CLBs. Double lines

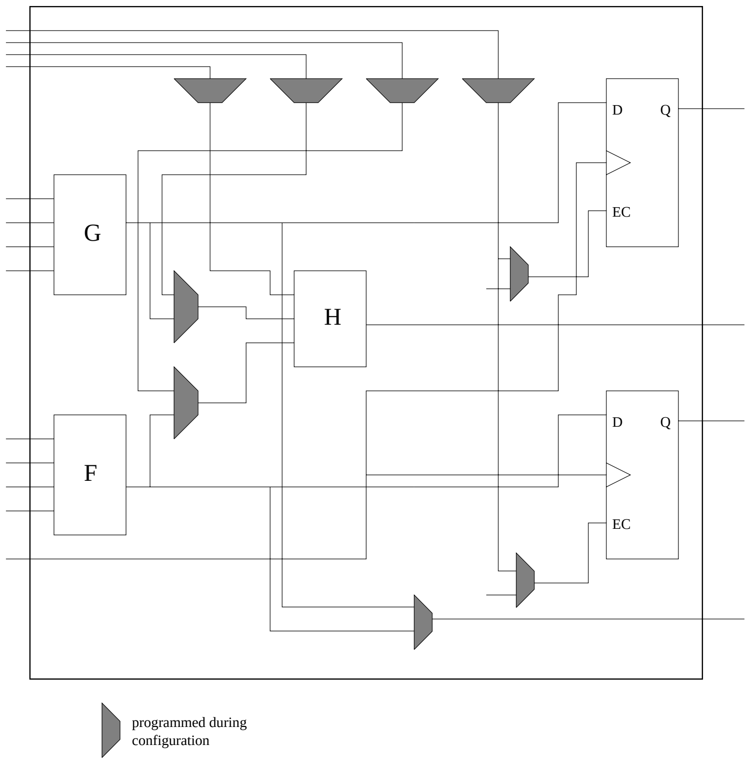

Figure 6.4: CLB structure (simplified)

stretch over two CLBs. For a detailed description of the routing structure inside the XILINX FPGA, please refer to [47]. Table 6.1 points out the difference in routing resources between the E and EX family of devices. As can be seen, the EX and consequently the XL family provide much more freedom to route data between CLBs. Choosing these devices (ones with more routing resources), proved to be necessary since the place and route tools had a difficult time routing even with the XL family.

	XC4000E		XC4000EX/XL	
	Vertical	Horizontal	Vertical	Horizontal
Singles	8	8	8	8
Doubles	4	4	4	4
Quads	0	0	12	12
Longlines	6	6	10	6
Direct Connects	0	0	2	2
Globals	4	0	8	0
Carry Logic	2	0	1	0
Total	24	18	45	32

Table 6.1: Routing per CLB in XC4000 devices

6.7 Serial vs. Parallel Computation Consideration

Figure 6.5 shows the general design architecture. In this graphic, the control architecture is shown as a series of multiplexers and signals used to enable individual components. Thus the control mechanism is responsible for routing the correct signal to and from the processing elements (PEs), asserting the necessary registers or issuing the correct RAM addresses, and enabling PEs for certain operations. The PEs shown in the figure are enabled when necessary to process data introduced to them. Please note that this figure supports only serial executions of every operation we wish to implement because the output of the multiplexers only asserts one datapath. It is easily seen with this graphic that implementing a parallel architecture would require much wider data buses requiring increased routing. The routing that has to be accomplished to implement the system in Figure 6.5 is quite involved since each register is $n \cdot m$ bits wide. As a final note, it is important to stress that this graphic shows a very general view of a digital system and can be used to implement virtually any computation. The purpose of this is to help visualize the different parts

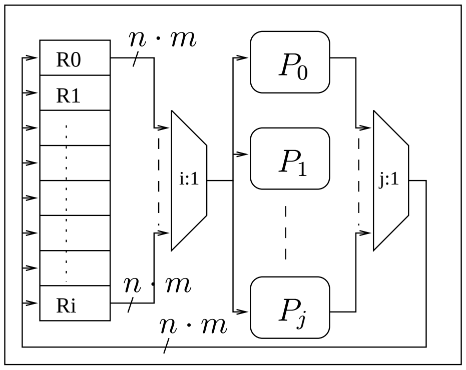

Figure 6.5: Top level design view

that constitute a system, even though the final design may look quite different from this figure especially since a hardware description language (HDL) will be used to realize it. Chapters 7 through 10 describe a particular elliptic curve implementation using projective coordinates. These chapters will provide the reader with a detailed description of our design.

Chapter 7

Design Overview

Our implementation had the goal of performing a complete EC point multiplication over $GF((2^n)^m)$ with $n \cdot m \geq 130$. One major advantage of using an FPGA as the target hardware is that the entire architecture can simply be reconfigured for different values of n and m. We re-introduce the graphic presented in Section 4.2 in the context of hardware implementation. Figure 7.1 shows the partitioning of the design into three levels. From the discussion in Section 4.2 it follows that the entire "crypto engine" can be divided hierarchically into three levels.

- **The Arithmetic Level** describes the processing elements that perform Galois field arithmetic. This is described in Chapter 8.
- **The Group Operation Level** combines the arithmetic modules as well as storage element(s) and control mechanism into a system architecture that realizes a single point addition or doubling. This is described in Chapter 9.
- **The Encryption Level** defines the I/O interface, control sequences and initialization commands that result in the realization of the double-and-add algorithm.

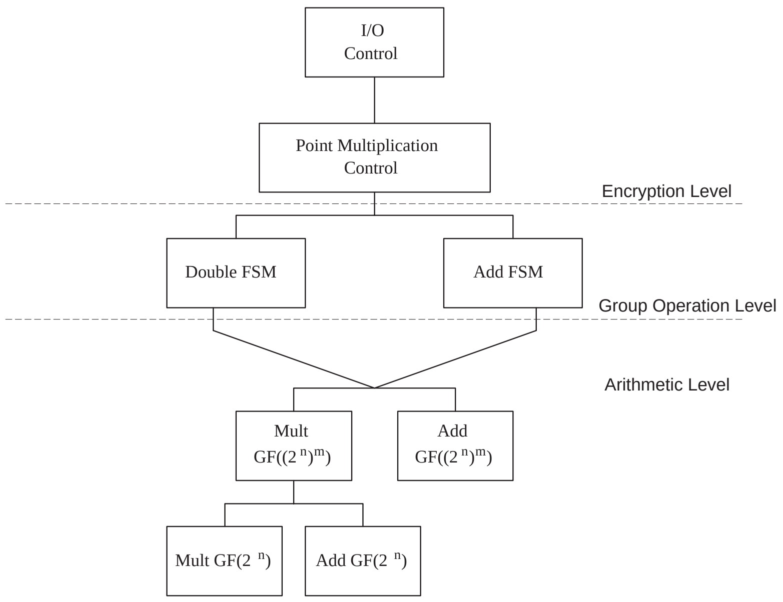

Figure 7.1: System hierarchy

This is described in Chapter 10.

The description of the digital design is divided into four chapters. In this chapter, we provide the general overview of the system putting emphasis on all static elements of the design. Static elements include all components that do not define the control mechanism of the design. Thus, storage elements and multiplexers are described in this chapter. The heart of the design is the control structure which is defined in three levels of hierarchy depicted in Figure 7.1. Because of the importance of this control mechanism, three chapters (Chapter 8, Chapter 9, Chapter 10) are dedicated to its description. The control engine can be thought of as the dynamic part of the design. These static components receive instructions from the control module and perform a certain task.

7.1 Design Overview

A general description of a digital system defines the processing elements, storage components, and control mechanism necessary to accomplish a certain task. Figure 7.2 shows the Register Transfer Level (RTL) view of the entire design. This is a coarse view of the system emphasizing the major components of the design. In addition, this diagram shows the I/O requirements as well as the datapath through the device.

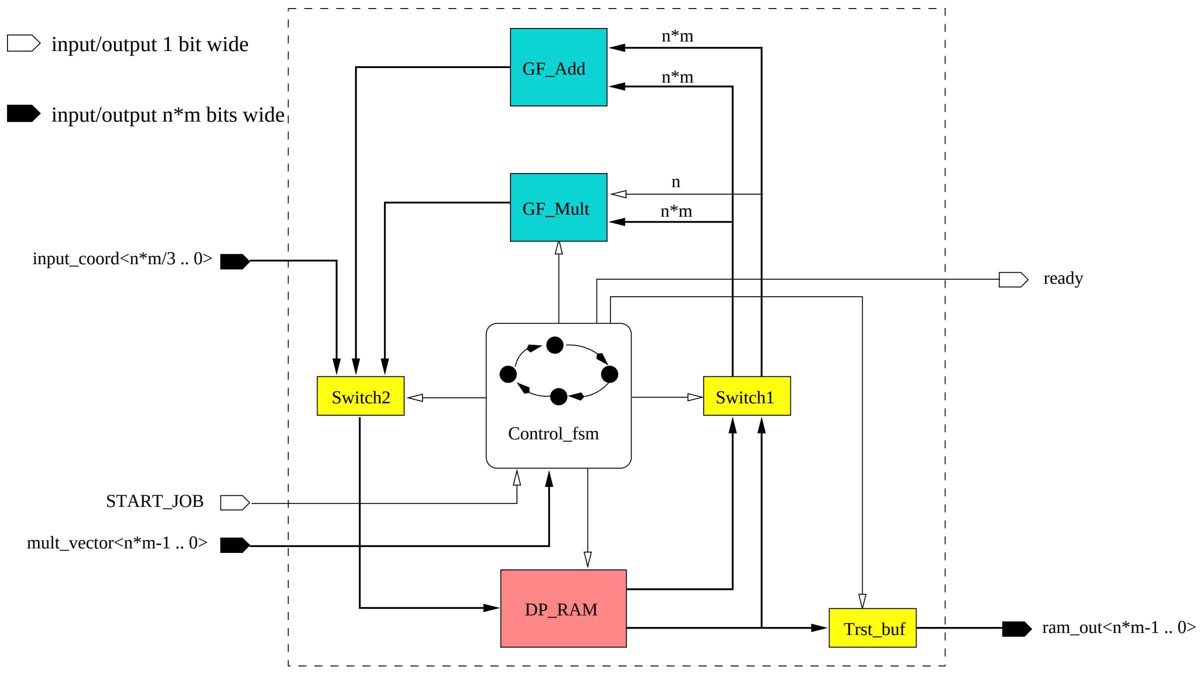

Figure 7.2: System diagram

Besides the control mechanism, the design is broken into three other parts. The storage elements such as RAM and registers are used to “latch” correct data between clock cycles; this is typically a synchronous operation. Switch boxes are purely combinatorial. They can be thought of as generalized multiplexers since they are used to select slices of data during specific clock cycles. Finally, the processing elements GF_Add and GF_Mult perform the arithmetic in $GF((2^n)^m)$. Point multiplication is achieved through a long series of instructions given by the control mechanism. The term "instructions" is used loosely here since the instruction is actually a bus of control signals routed to different blocks of the design. Putting all these signals together will generate an instruction specific to a unique clock cycle during point multiplication computation. Thus, the general approach of computing a new point is to provide correct data elements to the correct processing blocks at the correct time (right side of Figure 7.2). Conversely, a computed intermediate result is routed back to the RAM and written to the correct address at the correct time (left side of Figure 7.2). The tri-state buffer shown in the figure is used at the completion of point multiplication to route the result to output pins.

7.2 Storage

There are three types of storage elements in this design, two of which are part of the hybrid multiplier. The serial multiplier is a linear feedback shift register. It has to store intermediate results during each clock cycle of its operation. Consequently, some type of storage is necessary. In particular, registers without clear/preset are used to store the first operand in its entirety. This is the $V(x)$ operand shown in Figure 3.1. The clear/preset line is not needed here since these registers are always loaded with $V(x)$. However, the registers holding the result ($W(x)$ in Figure 3.1), have to be reset to all zeros before every serial multiplication is initiated. This is done while the $V(x)$ is being loaded to the upper register. A control signal, `load_gfm`, is asserted by the control architecture such that the result register can be cleared asynchronously.

The third storage element is the large RAM bank. This component stores up to

16 elements (address width of four bits), with each element having $n \cdot m$ bits. For instance, for the $GF((2^8)^{21})$ implementation 1688 bits of data can be stored in the RAM element. Furthermore the RAM bank is dual ported to accommodate concurrent read and write cycles. With this feature, it is possible to write the result of current computation while pre-fetching the first operand for the next operation. Consequently, one clock cycle is saved on every operation. Memory access from the PEs point of view is described later in this chapter. Here, only the general read/write cycle is shown in Figure 7.3. DPO and SPO are the two output busses that provide

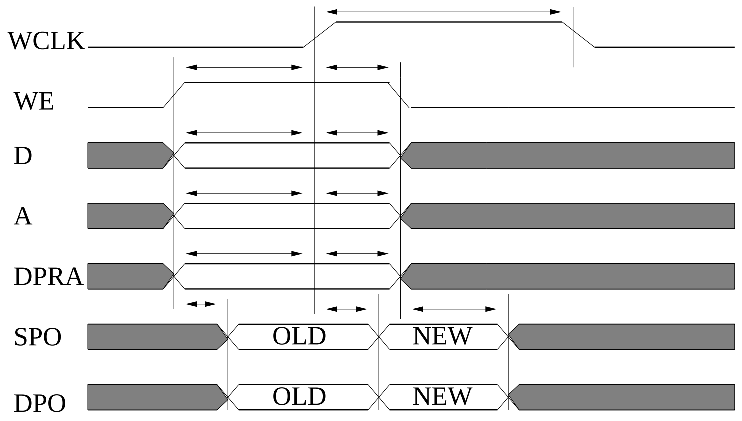

Figure 7.3: Read/Write cycle for dual ported RAM

data to the PEs. DRPA is the address for DPO and A is the address for SPO (single port output). DPO can be used to read or write resulting in the possibility of having two concurrent reads (as is done for the add operation) or a read and a write (as is done for the serial multiply operation).

7.3 Data/Address Select

Two switches are utilized to select data on its way to or from the processing elements. An additional switch is implemented to select instructions between Add_FSM and Double_FSM (described in the next chapter). Switches 2 and 3 function just as multiplexers but with the added complexity of some signal re-mapping. Switch 3, on the other hand has some added functionality depending on the current operation. The output vectors are provided with either one of the input vectors. If addition is performed, OUT1 receives IN1 and OUT2 receives IN2. Thus the switch just passes

	FSM Operation	
Output	GF_Add	GF_Mult
OUT1$[0 \ldots slice-1]$	IN1$[0 \ldots slice-1]$	IN1$[index \ldots index+slice-1]$
OUT1$[slice \ldots n \cdot m]$	IN1$[slice \ldots n \cdot m]$	XXX...XXX
OUT2$[0 \ldots n \cdot m]$	IN2$[0 \ldots n \cdot m]$	IN1$[0 \ldots n \cdot m]$

Table 7.1: Operation of Switch 1

the vectors through in their entirety. During serial multiplication however, the switch routes the entire IN1 to OUT2 on the first clock cycle and a "slice" of IN1 to OUT1 during the remaining cycles. Depending on the counter value provided by the control mechanism, a different slice is passed to the multiplier through OUT1. Table 7.1 summarizes this behavior.

Switch 2 is located after the processing elements. The function of this component is to select one out of three vectors. Two of the input vectors are $n \cdot m$ bits wide while the third one is $(n \cdot m) \div 3$ bits wide. These inputs are selected according to the type of operation that is being performed. During the load stage of the computation, Switch 2 passes the input vector so that the initial curve coefficients can be loaded into the appropriate locations. The input vector is replicated three time inside Switch 2 and a data bus is built of the same width as the output $(m \cdot n)$. Table 7.2 defines the functionality of Switch 2. Since the load operation only loads 1/3 of entire data

	FSM Operation		
Output	Load	GF_Add Write	GF_Mult Write
OUT[block1]	IN1[block1]	Add_result[block1]	Mult_result[block1]
OUT[block2]	IN1[block1]	Add_result[block2]	Mult_result[block2]
OUT[block3]	IN1[block1]	Add_result[block3]	Mult_result[block3]

Table 7.2: Operation of Switch 2

length for each clock cycle, a mechanism for selecting which RAM element to write to is required. This is done with a write_enable vector of width $(m \cdot n)$. Thus, during the first clock cycle of data load, 1/3 third of the bits in the write_enable vector are asserted. Consequently, only one third of the RAM width is written to and the rest of the bits are irrelevant. Thus replicating the bits for the entire width inside Switch 2 ensures that data gets written to the correct block of RAM without any additional logic. The other solution would require extra logic that would assign the input coordinates to the correct portion of the data bus for each clock cycle of the load stage. With the approach outlined in Table 7.2, no additional logic is necessary at the expense of extra routing resources. However, since the additional routing is confined to only this component, single lines can be utilized when the design is mapped into a particular device leaving the other resources free.

Switch 3 is internal to the control mechanism. It's purpose is to select instructions streams coming in from the I/O, DOUBLE, and ADD state machines. The I/O state machine control is enabled during the load and unload stages of computation. This control is only enabled twice per point computation as coordinates have to be loaded and unloaded to/from the engine only once. Table 7.3 summarizes the operation of

	FSM Operation			
Output	Load	Unload	DOUBLE	ADD
RAM_WRT_EN	block	none	all	all
RAM_WRT_ADDR	LD_WRT	none	DBL_WRT	ADD_WRT
RAM_RD_ADDR	none	UNLD_RD	DBL_RD	ADD_RD

Table 7.3: Operation of Switch 3

Switch 3 and shows the control signals that are multiplexed. The RAM_WRT_EN signal is a bus which is $(m \cdot n)$ bits wide. During the write operation, it is entirely enabled for DOUBLE and ADD sequences. For the load sequence, this vector is enabled on a per block basis. Double and add control is enabled during computations depending on the current operation. As mentioned previously, repeated double and add operations are performed to achieve point multiplication. Thus engine control signals are effectively mapped to the double or add control.

7.4 Datapath Requirements

This section outlines combinatorial delays and resulting clock requirements of individual blocks of the design. Combinatorial components have to be considered in the context of datapath and absolute delay between clocked components. In addition, clocked resources determine the clock period required to operate these individual component. Thus, to thoroughly analyze the data path, delays associated with data movements and control sequencing have to be known. Finally, timing analysis is performed to estimate minimum clock period of the system after datapath delays are derived.

7.4.1 Combinatorial Components

Combinatorial components include all the switches and the addition module. The maximum path delay through Switch 1 has to be considered when computing the minimum clock period because this component is in the path between the RAM module and the multiplier. Switch 3 multiplexes the control signals from the double and add state machines. As a result, this component also contributes a delay that defines the clock period. After the result is computed, it is passed through Switch 2. This switch is also controlled by the state machine. However, since the multiplication result is not ready until the next falling edge of the clock, the control signal selecting the addition or multiplication result has nearly half a period to settle. Thus, by the time valid data from the processing elements arrives, Switch 2 is already selecting the correct input. Consequently, only the delay associated with routing the result through this component has to be considered.

7.4.2 Clocked Resources

Clocked resources in the design include the multiplier, the control logic, and the RAM module. Furthermore, the control logic is partitioned into three state machines. Thus the slowest state machine will determine the clock of the entire control module. The RAM bank is structured with a slice architecture resulting in a concurrent memory access to all slices. The parallel load of data and address assures that each memory element is provided with data at the same time. Thus any delay associated with this component is due to the setup and hold parameters of a RAM slice specified by the manufacturer. In this case, XILINX is the manufacturer since the design utilizes a soft macro. Using this macro guarantees the necessary clock period specified by XILINX. The multiplier is also based on a slice architecture but with a feedback path.

Consequently, the clock period depends on the array size making this component implementation specific.

7.4.3 System Timing Analysis

The coarse description of the EC digital system in Figure 7.2 shows the flow of data inside the architecture. The data flows counter clockwise originating at the RAM element. After passing through combinatorial delay of Switch 1, data enters one of two processing elements. Once the result is computed, it is moved through Switch 2 and back to the RAM bank. This is the basic data flow during double or add computation. During the load stage the input coordinates are routed to the RAM bank through Switch 1. Thus only the combinatorial delay through this block effects the data arrival time. Once the computation is completed (double and add operations have been sequenced), the result located in the top three RAM locations is moved to the output pins through a tri-state buffer. Consequently, the delay through the tri-state gates is taken into account when specifying the hold time for the output pins of the device.

To estimate the system clock speed two steps are necessary. First, all combinatorial delays through each synchronous path have to be summed and the minimum clock period for each path has to be derived. Consequently due to the synchronous nature of this implementation, the path with the highest period determines the system clock speed.

In our implementation, three distinct paths can be derived. The first path realizes the load operation. As mentioned previously, combinatorial delay associated with this path includes Switch 1 in addition to the pad delays. The pad delays can be ignored as the synthesis and place and route tools can schedule the clock according to constraints. These constraints allow us to specify the setup time associated with pad delay. Thus, the load path delay cab be approximated to be:

$$\text{load\_path} = t_{ds1} || t_{ds3}$$

where t_{ds1} is associated with arrival time of data and t_{ds3} determines the arrival time of control signals to the RAM module. The symbol || is used to show parallel operation. Thus, the longest delay determines the path.

The second possible path is realized during the unload operation. This occurs after point multiplication is completed and it is determined solely by the tri-state buffers and the output enable signal controlled by the state machine. This path has the following propagation delay:

$$\text{unload\_path} = t_{dtrst} || t_{den}$$

where t_{dtrst} is the delay of the buffer and t_{den} is the time required by the control state machine to assert the enable signal.

The last synchronous path that can be analyzed is derived during the arithmetic computation. This path includes either of the processing elements. When the addition module is used, two clock cycles are allocated for result to settle before it is written to the RAM bank. Consequently, the combinatorial delay associated with this operation is one half of all delays in the data path. Thus the delay during addition is:

$$\text{add\_path} = (t_{ds1} + t_{dadd} + t_{ds2}) \div 2 || t_{ds3}$$

When considering the data path associated with multiplication, both clock edges have to be analyzed. More specifically, data has to have enough time to get to the multiplier before the falling edge of the clock and enough time to get to the RAM module before the rising edge of the clock. Thus the clock cycle can be divided into two parts:

$$\text{mult\_path\_high} = t_{ds1} + t_{ds3}||t_{ds3} = t_{ds1} + t_{ds3}$$

$$\text{mult\_path\_low} = t_{ds2}||t_{ds3}$$

The summation of both of these results will result in the data path delay for the multiplication operation:

$$\text{mult\_path\_all} = \text{mult\_path\_high} + \text{mult\_path\_low}$$

Finally, the system clock is determined by the longest combinatorial delay:

$$\text{system\_path} = \text{mult\_path\_all}\ ||\text{add\_path}\ ||\text{load\_path}||\text{unload\_path}$$

The above theoretical analysis serves as a guideline for timing constraints that can be issued during the synthesis and/or place and route stages of our implementation. Assuming that all other clocked resources can run at this clock speed this estimate provides a good measure of the system clock. This analysis does not include setup and hold time requirements of clocked components as these values are highly dependent on the place and route tools used and the optimization constraints applied.

Chapter 8

Arithmetic Level

As described previously, multiplication and addition is performed in $GF((2^n)^m)$. This section defines the control architecture that was developed to realize these function. Multiplication is done slice serially and addition can be performed in one clock cycle. With each architecture, a low level state machine is described. From the hierarchal view the Galois field operations described in this section form the arithmetic level. These function blocks together with the corresponding control structure compose the next higher level which is the double and add or group operation level. The group operation level is described in Chapter 9 and the encryption level is described in Chapter 10. The higher levels ensure that the arithmetic operations are performed in the correct sequence to realize point multiplication.

8.1 Multiplication in Hardware

Galois field multiplication is accomplished through a slice architecture described in Section 4.2. As stated initially, one multiplication requires $m + 1$ clock cycles. That's

m clock cycles for the multiplication and an additional cycle to write the result into memory. The two operands required for multiplication are read from memory as shown in Figure 8.1. The first operand that has to be loaded in to the internal registers of the multiplier is read during the write cycle of the previous operation. While this operand is registered, the registers holding the result are cleared so that the new multiplication sequence can start. The second operand is read from memory

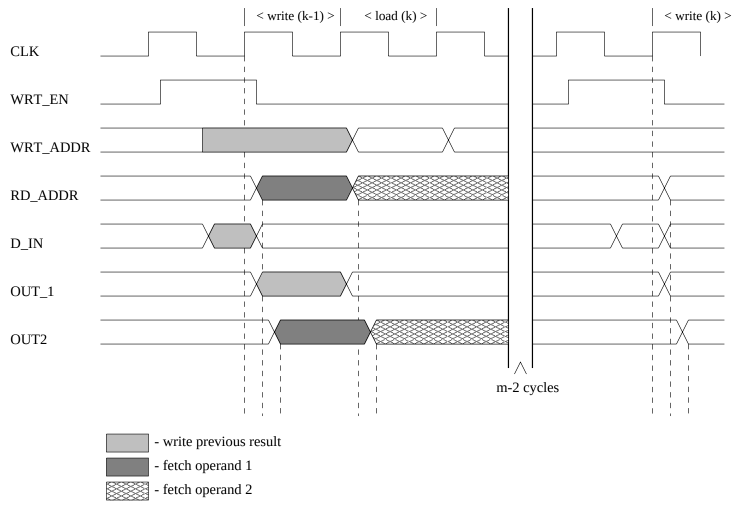

Figure 8.1: Memory access for multiplication

during the first clock cycle of multiplication stage. This operand is then introduced to the multiplier one slice per clock cycle. Switch 1 shown in Figure 7.2 is responsible for selecting the correct slice and routing it to the multiplier. The timing diagram in Figure 8.1 shows the relative timing delay associated with Galois field multiplication. From a control point of view, the state machine for this serial multiplication requires three states. This state machine is the lowest level hierarchy of the control mechanism. In Chapter 9, the group operation control structure is described.

This means that the underlying arithmetic control is at a lower level and is described in this section. Figure 8.2 shows the control signals associated with each state of

Galois field multiplication. The load stage reads the second operand from memory (the first operand is read during write stage of previous computation). The calculate stage loops through $slice - 1$ clock cycles and controls the output of Switch 1. In

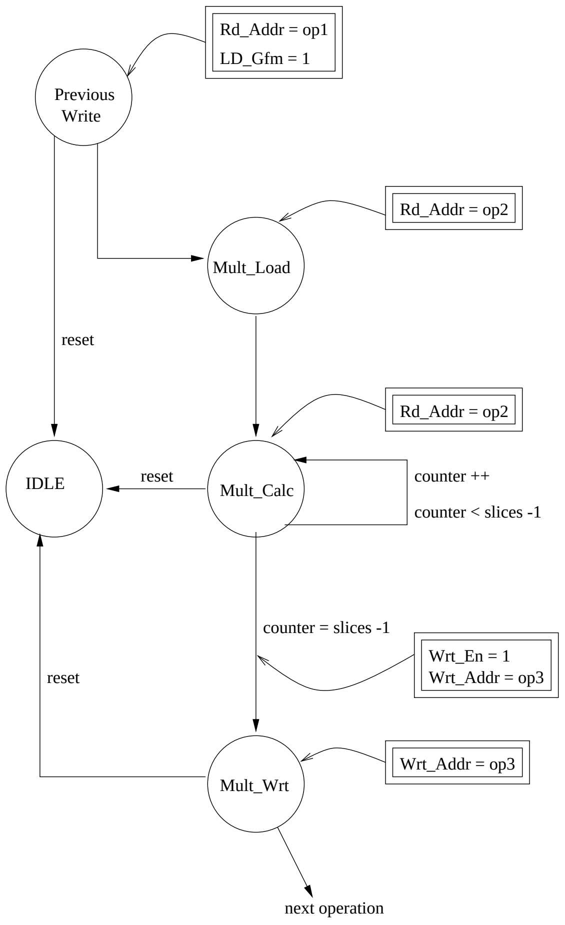

Figure 8.2: Arithmetic level FSM for Galois field multiplication

the write stage of the multiplication, the result is written to a third memory location and a new operand is read for the next computation. After the write, the state multiplication sequence is completed and a new operation can begin immediately.

One important design characteristic is the operation of the multiplier relative to the operation of the control structure. Because of the combinatorial delay of control signals, the multiplier cannot start its operation until the correct data is ready to be latched in. Thus the multiplier operates on the falling edge of system clock while the rest of the system runs of the rising edge. The combinatorial delay associated with reading the correct memory location and routing this data to the multiplier has to be less than half the system clock cycle unless the clock has a duty cycle greater than 0.5. The result of serial multiplication is ready half a clock cycle before the rising edge. Consequently, Switch 2 has to deliver this result in less the half a clock cycle. This approach introduces a latency of half clock cycle during multiplication as opposed to an delay of entire clock cycle that would be necessary if the multiplier was clocked on the rising edge.

8.2 Addition in Hardware

Since addition can be done with a simple bitwise XOR function only one clock cycle is necessary to compute the result. Figure 8.3 shows the memory access for a Galois field addition operation. As shown, both of the operands are read at the same time and passed through Switch 1 to the adder. During the same clock cycle the result is computed. The additional clock cycle is necessary to write the result back to memory and to read one operand if multiplication is the next operation. Addition is purely combinatorial and latching of intermediate data is not necessary. Thus addition requires two clock cycles which is relatively fast in comparison to Galois field multiplication. Because of the low complexity of this function, the state machine controlling this process is rather simple. This state machine, shown if Figure 8.4, only has two states. In the load state two memory locations are addressed simultaneously so that both of the operands can be introduces to the adder at the same time. The next state writes the result of the addition into memory and performs read operations for the next computation. No counter is necessary for this sequence as each state only

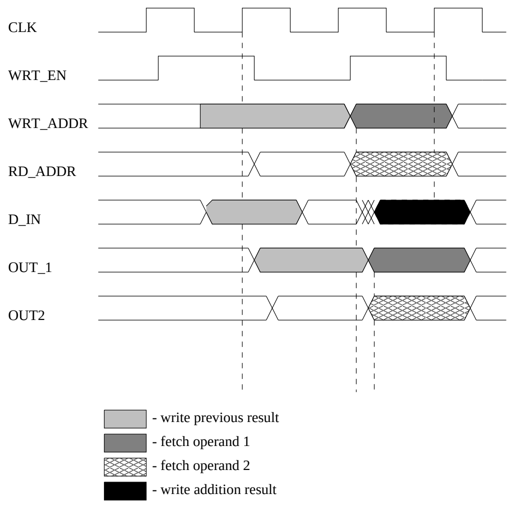

Figure 8.3: Memory access for addition

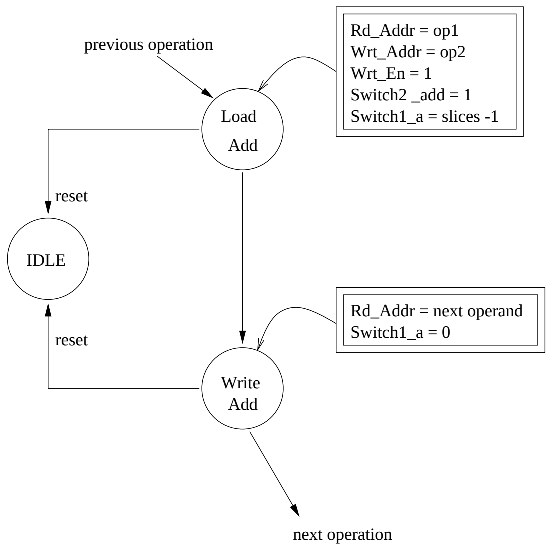

Figure 8.4: Arithmetic level FSM for GF addition

lasts one clock period. This simplifies the control structure for the addition operation. After the completion of this operation, a new addition or multiplication can begin on the next clock.

8.3 State Machine Encoding

The two small state machines that realize addition and multiplication in $GF((2^n)^m)$ are embedded in the higher level state machine. More specifically, the two state machines that describe Group Operation Level perform either addition or multiplication in ordered sequence. Each addition and multiplication is defined as shown in Figure 8.4 and Figure 8.2 respectively. Before synthesis takes place, the hierarchal state machine is flattened. Consequently, state encoding for the Arithmetic Level and the Group Operation Level state machines is the same. The encoding schemes that we implemented in these state machines are described in Chapter 9.

Chapter 9

Group Operation Level

This chapter describes the design of the control mechanism that allows for a sequential computation of the third point on the elliptic curve given two points. The process is divided into two distinct operations. One of them realizes point doubling ($P = Q$) and the other implements point addition ($P \neq Q$). It is essential to remember that the set of operations for point doubling is significantly different from operations realizing point addition. For this reason, the control mechanism can be viewed as having two separate sequence schedulers. These schedulers are controlled by a third state machine which defines the Encryption Level. As shown in Figure 9.1, the control mechanism exhibits a hierarchal structure. I/O state machine is the root of all control issuing commands to either double or add state machines. Furthermore, the I/O FSM provides control signals to the switches. This state machine is described in Chapter 10. These control signals are used by the switches to select the appropriate control between double or add control signals.

The chapter is divided into two sections. Section 9.1 describes the control sequence for the point double operation and Section 9.2 outlines the control sequence for the

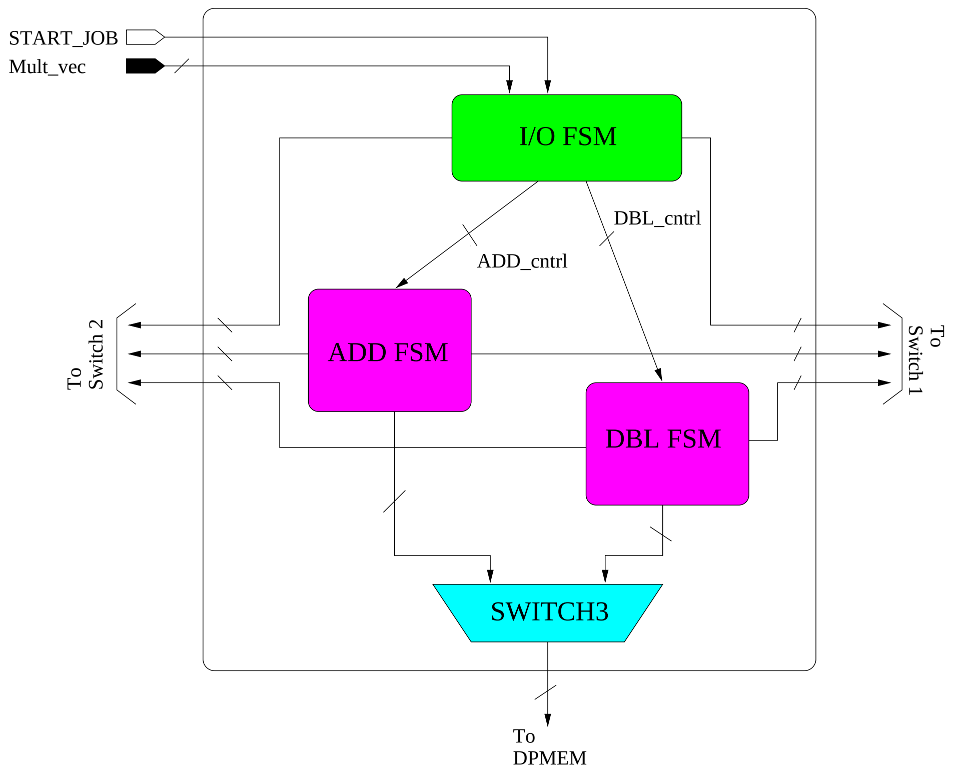

Figure 9.1: Internal architecture of control

point add operation. Both sections are further divided describing the precomputation and computations stages of each sequence.

9.1 Double Sequence Operations

As previously mentioned the set of operations that have to be performed to realize point doubling are as follows:

$$
\begin{aligned}
x_k &= AB \\
y_k &= x_{k-1}^4 A + B(x_{k-1}^2 + y_{k-1}z_{k-1} + A) \\
z_k &= A^3
\end{aligned}
$$

where $A = (x_{k-1}z_{k-1})$, $B = (a_6 z_{k-1}^4 + x_{k-1}^4)$. These operations can be broken down into two stages. The first stage performs precomputations of intermediate values. These values are then passed to the calculation stage that computes the three coordinates x_k, y_k, z_k using the intermediate values. The term 'passing' is used loosely here since the actual data does not have to move when entering the computation stage. Furthermore, the actual design of point double or point add sequencer is not realized in two separate stages. The description given here distinguishes between these stages only for clarity reasons in that the reader will have an easier time following the mapping between the given equations and the corresponding operations. Thus the precomputation stage for point double realizes the intermediate value A and B and computation stage uses these results to compute the desired point. The resulting point coordinates are stored in memory locations 0h-2h and can be used again to perform either point doubling or addition on the next clock cycle.

9.1.1 Precomputation Stage

The precomputation stage is responsible for computing the value of A and B as well as two more intermediate values x_{k-1}^4, and x_{k-1}^2 that are used in the computation stage. The value a_6 which is one of the curve parameters, is stored in the lower memory location and is read only in the precomputation stage of the point double algorithm.

Since the calculation of B requires x_{k-1}^4 and the calculation of x_{k-1}^4 requires x_{k-1}^2, the values have to be computed sequentially. Storing these values in lower memory locations will help in the next stage as they will not have to be recomputed. This is a classic example of time space trade-off. In this case, more space is sacrificed so that the number of operations can be reduced. The abundance of memory locations makes this choice easier and all intermediate results that are necessary in the computation stage are preserved.

Initial State:

The initial values assumed prior to the k-th double computation are as follows:

$$
\begin{aligned}
\text{RAM0} &= x_{k-1} \\
\text{RAM1} &= y_{k-1} \\
\text{RAM2} &= z_{k-1} \\
\text{RAM3} &= x_0 \\
\text{RAM4} &= y_0 \\
\text{RAM5} - \text{RAM13} &= \text{empty} \\
\text{RAM14} &= a_6 \\
\text{RAM15} &= a_2
\end{aligned}
$$

Double Precomputation Sequence:

Depending on the number of data dependencies between each computation, the sequence can be scheduled in such fashion so that memory writes occur concurrently with some computation. This will save one clock cycle for each operation. The simplest way of ensuring that memory writes can be scheduled concurrently with the next computation is to make sure that the result from the previous computation is not used in the next one. Thus, for a point double precomputation stage, the sequence shown in Table 9.1 was developed.

Figure 9.2 shows the corresponding memory allocation for the sequences outlined above. In state 7 of the precomputation stage, the last computation is performed as shown in the figure depicting the active registers.

#	*Operation*	*Result*	*Write*
1.	RAM8 = (RAM2)^2	z_{k-1}^2	none
2.	RAM5 = (RAM0)^2	x_{k-1}^2	RAM8
3.	RAM6 = (RAM8)^2	z_{k-1}^4	RAM5
4.	RAM8 = (RAM5)^2	x_{k-1}^4	RAM6
5.	RAM7 = RAM14 · RAM6	$z_{k-1}^4 \cdot a_6$	RAM8
6.	RAM9 = RAM0 · RAM2	$x_{k-1} \cdot y_{k-1} = A$	RAM7
7.	RAM7 = RAM7 + RAM8	$z_{k-1}^4 \cdot a_6 + x_{k-1}^4 = B$	RAM9

Table 9.1: Double_fsm precomputation sequence

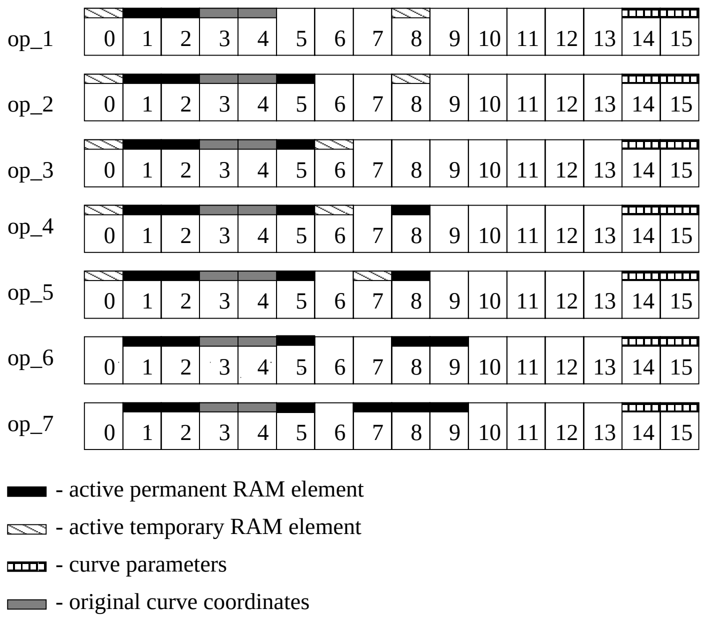
op_1
op_2
op_3
op_4
op_5
op_6
op_7
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
- active permanent RAM element
- active temporary RAM element
- curve parameters
- original curve coordinates

Figure 9.2: Memory allocation table for double precomputation stage

Check: By quick substitution it is possible to verify that the result after precomputation stage is indeed the desired one.

$$\begin{aligned}
\text{RAM9} = \text{RAM0} \cdot \text{RAM2} \quad &= x_{k-1} \cdot z_{k-1} = A \ \surd \\
\text{RAM8} = (\text{RAM5})^2 \quad &= ((\text{RAM0})^2)^2 = \text{RAM0}^4 = x_{k-1}^4 \ \surd \\
\text{RAM5} = (\text{RAM0})^2 \quad &= x_{k-1}^2 \ \surd \\
\text{RAM7} = \text{RAM7} + \text{RAM8} \quad &= \text{RAM14} \cdot \text{RAM6} + \text{RAM5}^2 \\
&= \text{RAM14} \cdot \text{RAM8}^2 + (\text{RAM0}^2)^2 \\
&= \text{RAM14} \cdot (\text{RAM2}^2)^2 + (\text{RAM0}^2)^2 \\
&= a_6 \cdot z_{k-1}^4 + x_{k-1}^4 = B \ \surd
\end{aligned}$$

Final State:

The state of memory after the last operation in the precomputation stage is passed to the computation stage. The active registers contain the following information:

$$\begin{aligned}
\text{RAM0} &= x_{k-1} \\
\text{RAM1} &= y_{k-1} \\
\text{RAM2} &= z_{k-1} \\
\text{RAM3} &= x_0 \\
\text{RAM4} &= y_0 \\
\text{RAM5} &= x_{k-1}^2 \\
\text{RAM7} &= B \\
\text{RAM8} &= x_{k-1}^4 \\
\text{RAM9} &= A \\
\text{RAM10} - \text{RAM13} &= \text{empty} \\
\text{RAM14} &= a_6 \\
\text{RAM15} &= a_2
\end{aligned}$$

Note that the computation stage does not need the value of x_{k-1}, since x_{k-1}^2 and x_{k-1}^4 are already computed. This frees that memory location for the use in the computation stage. It is also important to realize that RAM3 and RAM4 were not written to or read from during this stage. This is because point doubling operation does not require the original coordinates that are stored at these memory locations.

9.1.2 Computation Stage

At the end of this stage the following results should be calculated:

$$\begin{aligned} x_k &= AB \\ y_k &= x_{k-1}^4 A + B(x_{k-1}^2 + y_{k-1}z_{k-1} + A) \\ z_k &= A^3 \end{aligned}$$

where A and B are the results of precomputation calculations.

<u>Initial State:</u>

The initial state of this stage is the final state of the precomputation stage:

$$\begin{aligned} &\text{RAM0} = x_{k-1} \\ &\text{RAM1} = y_{k-1} \\ &\text{RAM2} = z_{k-1} \\ &\text{RAM3} = x_0 \\ &\text{RAM4} = y_0 \\ &\text{RAM5} = x_{k-1}^2 \\ &\text{RAM7} = B \\ &\text{RAM8} = x_{k-1}^4 \\ &\text{RAM9} = A \\ &\text{RAM10} - \text{RAM13} = \text{empty} \\ &\text{RAM14} = a_6 \\ &\text{RAM15} = a_2 \end{aligned}$$

At this point four memory locations are still free and can be used in the computation stage. The intermediate values computed in the previous stage are depicted as temporary memory locations in Figure 9.3. Once these values serve their purpose, the memory location is freed and ready to be written again. Locations marked as permanent in the memory allocation tables are the final results for that particular stage. Thus at the end of the double sequence, permanent locations are RAM0, RAM1, and RAM2 containing the k-th point coordinates.

Computation Sequence:

Table 9.2 outlines the sequence for the computations stage of the double operation.

#	*Operation*	*Result*	*Write*
1.	RAM6 = RAM1 · RAM2	$y_{k-1}z_{k-1}$	RAM7†
2.	RAM5 = RAM9 + RAM5	$x_{k-1}^2 + A$	RAM6
3.	RAM1 = (RAM9)^2	A^2	RAM5
4.	RAM6 = RAM5 + RAM6	$x_{k-1}^2 + y_{k-1}z_{k-1} + A$	RAM1
5.	RAM2 = RAM1 · RAM9	z_k	RAM6
6.	RAM1 = RAM6 · RAM7	$B(x_{k-1}^2 + y_{k-1}z_{k-1} + A)$	RAM2
7.	RAM6 = RAM8 · RAM9	$A \cdot x_{k-1}^4$	RAM1
8.	RAM0 = RAM7 · RAM9	x_k	RAM6
9.	RAM1 = RAM6 + RAM1	y_k	RAM0
10.			RAM1

† - last result from precomputation stage

Table 9.2: Double_fsm computation sequence

At the end of this sequence, the last memory write is performed to store the result for y_k. Seeing as how the computation of y_k is quite complicated, this result is obtained last. Thus the total number of states required for this computation is ten, and the number of clock cycles necessary for each state depends on the type of operation assigned to that state. For example, state 1 involves multiplication therefore it requires multiple clock cycles. However, state 9 only performs addition of polynomials that can be done with a bitwise XOR function within one clock cycle.

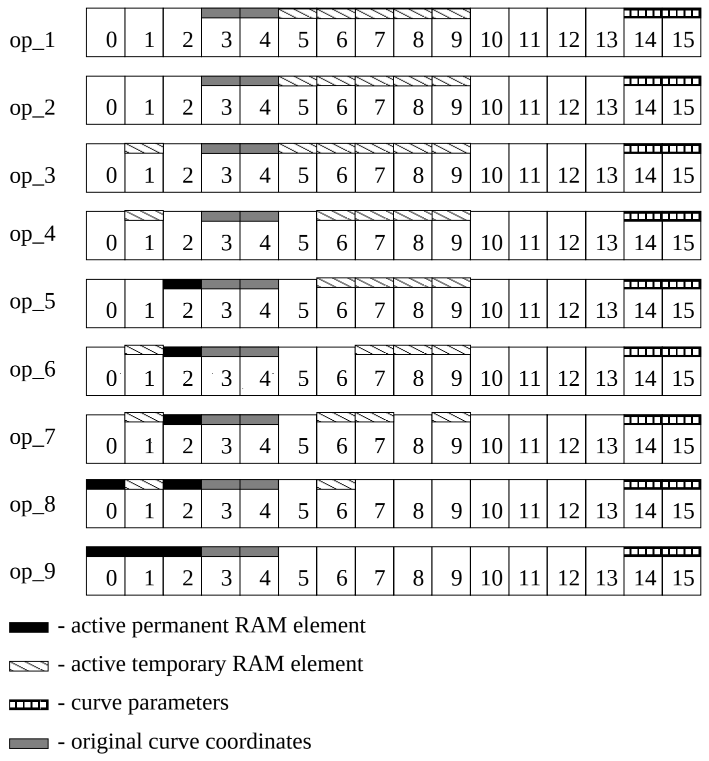

Figure 9.3: Memory allocation for double computation stage

Figure 9.3 shows the memory usage for the corresponding sequences outlined above. In state 9 of the computation stage, the last operation is performed as shown in the figure depicting the active registers at the end of the sequence.

Check: Checking the arithmetic again shows that at the end of this stage equations for point doubling have been satisfied:

$$\begin{aligned}
\text{RAM0} = \text{RAM7} \cdot \text{RAM9} \quad &= A \cdot B = x_k \ \surd \\
\text{RAM1} = \text{RAM6} + \text{RAM1} \quad &= \text{RAM8} \cdot \text{RAM9} + \text{RAM6} \cdot \text{RAM7} \\
&= \text{RAM8} \cdot \text{RAM9} + (\text{RAM6} + \text{RAM5})\text{RAM7} \\
&= \text{RAM8} \cdot \text{RAM9} + (\text{RAM1} \cdot \text{RAM2} + \text{RAM9} + \text{RAM5})\text{RAM7} \\
&= x_{k-1}^4 A + B(y_{k-1} z_{k-1} + A + x_{k-1}^2) = y_k \ \surd \\
\text{RAM2} = \text{RAM1} \cdot \text{RAM9} \quad &= \text{RAM9}^2 \cdot \text{RAM9} \\
&= \text{RAM9}^3 = A^3 = z_k \ \surd
\end{aligned}$$

<u>Final State:</u>

The state of the registers after the last operation in the computation stage is the desired result for point doubling with the three coordinates x_k, y_k, and z_k in RAM0, RAM1, and RAM2 respectively.

$$\begin{aligned}
\text{RAM0} &= x_k \\
\text{RAM1} &= y_k \\
\text{RAM2} &= z_k \\
\text{RAM3} &= x_0 \\
\text{RAM4} &= y_0 \\
\text{RAM5} - \text{RAM13} &= \text{empty} \\
\text{RAM14} &= a_6 \\
\text{RAM15} &= a_2
\end{aligned}$$

Point doubling can be performed immediately after a previous point double since the registers do not have to loaded with new values. Point addition can also be started immediately following this stage as the memory locations containing original coordinates were not overwritten.

9.1.3 Summary

With the state machine sequence known for the precomputation and computation stage, it is possible to estimate the number of clock cycles required for the point doubling operation. First, general formulae are developed from which the number of clock cycles can be calculated in terms of the degree of the polynomial used. If a composite field architecture is used, the number of clock cycles will depend on the number of slices necessary for a certain polynomial degree and the clock period will depend on the degree of the subfield.

General Case

Storage Requirements: The maximum memory utilized is ten locations including the "permanent" RAM locations holding the curve parameters. The total memory required is $10 \cdot n \cdot m$ where n and m are the extension degrees of the composite field GF$((2^n)^m)$. For a secure elliptic curve system, $m \cdot n \approx 160$, thus a useful implementation requires $\approx 1.6kb$ of storage. This is a minimum requirement and in the implementation a memory block of depth 16 was used resulting in $16 \cdot n \cdot m$ bits. Thus roughly 2.56kb or 320 bytes of memory are implemented in the FPGA device.

Clock Cycles: In order to estimate the total number of clock cycles for point doubling, the number of field multiplications, squarings, and additions has to be computed. Furthermore, since the squaring operation is performed with multiplication, the number of clock cycles for the square operation are the same as for field multiplication. Thus, the precomputation stage requires the following operations:

$$\begin{aligned} &\#MULT + \#SQUARE = 6 \\ &\#ADD = 1 \end{aligned}$$

Similarly, the computation stage requires:

$$\#MULT + \#SQUARE = 6$$
$$\#ADD = 2$$
$$\#WRITE = 1$$

The last write cycle requires one clock cycle. Furthermore addition requires two clock cycles as described in Section 3.2 and according to Section 3.4, multiplication is computed in $m + 1$ clock cycles where m is the number of slices in the hybrid multiplier architecture. Thus, the total number of clock cycles for point doubling is

$$12 \cdot (m + 1) + 4 \cdot 2 + 1 = 12m + 21. \tag{9.1}$$

In this case m determines the number of slices of the composite field architecture, and n plays a major role in defining the minimum clock frequency of the serial multiplier.

Example GF$((2^8)^{21})$

The above formulae can now be used to estimate the time for one point doubling for a possible elliptic curve scheme. As an example, $\mathrm{GF}((2^8)^{21}) \equiv GF(2^{168})$ is used.

Storage Requirements: Once again, in theory, point doubling can be accomplished with $8 \cdot 21 \cdot 10 = 1.68$Kb of memory. That is 10 memory locations times the width of the point coordinate which is $8 \cdot 21$. When mapped into a XILINX FPGA, the use of dual-ported RAM is allowed for depths of power of two. Consequently the closest depth to 10 is 16 resulting in a memory bank composed of dual-ported blocks. According to [47], a single memory block (16x1 DPMEM) requires one CLB. Thus the entire memory block requires $n \cdot m$ CLBs.

Clock Cycles: The total number of clock cycles required to double one point on this curve is $12 \cdot 21 + 21 = 273$. Since a composite architecture is utilized, the field multiplication is reduced to a subfield operation. As a result, the minimum clock period for this design is the time necessary for one multiplication in the subfield GF(2^8). Previous research of the Galois Field multiplier on reconfigurable hardware [37], has shown that one subfield multiplication can be done in 31.1 ns. Thus, with a clock period of 60 ns (assuming 100% overhead for other combinatorial delay and net delays), the time required for one point doubling is $60ns \cdot 273 = 16.38\mu s$. This result is a rough estimate determined by a worst case scenario. By optimizing the square algorithm or introducing a variable clock, the frequency can be significantly improved resulting in a faster computation of point doubling.

9.2 Add Sequence Operations

Once again, the computation of the entire add sequence can be divided into the precomputation stage followed be the computation stage. The final result of these computations should yield the following:

$$\begin{aligned} x_k &= AD \\ y_k &= CD + A^2(Bx_{k-1} + Ay_{k-1}) \\ z_k &= A^3 z_{k-1} \end{aligned}$$

where $A = (x_0 z_{k-1} + x_{k-1})$, $B = (y_0 z_{k-1} + y_{k-1})$, $C = A + B$, and $D = A^2(A + a_2 z_{k-1}) + z_{k-1}BC$. For these computations, the curve parameter a_2 is accessed from memory location RAM15. Another important issue in point addition is the allocation of more memory. Since $P \neq Q$, two points are necessary to calculate the result with each point is described by three coordinates. It is important to note that only five coordinates are necessary since point add equations are optimized such that $z_0 = 1$.

However, since the soft macro available from the vendor of the FPGA device realizes memory of depth 16, extra memory locations are available.

9.2.1 Precomputation Stage

The precomputation stage computes all of the intermediate components that are A, B, C, D, and A^2.

Initial State:

The initial values needed for this stage are as follows:

$$\begin{aligned}
&\text{RAM0} = x_{k-1}\\
&\text{RAM1} = y_{k-1}\\
&\text{RAM2} = z_{k-1}\\
&\text{RAM3} = x_0\\
&\text{RAM4} = y_0\\
&\text{RAM5} - \text{RAM13} = \text{empty}\\
&\text{RAM14} = a_6\\
&\text{RAM15} = a_2
\end{aligned}$$

This algorithm requires one permanent value stored in memory location RAM15 and five coordinates that describe the two points to be added. As a result, $6 \cdot m \cdot n$ bits are stored before computation begins. These values are already in memory as a result of the previous computation or previous load. Therefore no additional clock cycles are required to load memory with initial coordinates.

Add Precomputation Sequence:

The data dependencies of the precomputation stage require a copy operation. The intermediate result C is computed by adding A to B. Furthermore the result of an addition operation is written back to one of the operands being added. Thus in order to add A to B a memory location has to be preloaded with one of the operand. Since both A and B are needed in the computation stage, both have to be preserved, and operand A has to be copied to the location where C will be stored.

The simplest way to copy one element from one memory location to another is to implement constructs in the control architecture to provide a feedback loop from data_out to data_in of the dual-ported RAM. This however requires additional routing resources which are limited in FPGA devices. Using this extra resources can be prevented thanks to the simple nature of the addition unit. As described earlier, addition in $GF(2)$ is accomplished with an XOR gate. Thus, copying an element can be done by adding it to the destination just as long as the destination is empty ($''0000\ldots000''$). To ensure that destination is empty another addition has to be performed. Mainly, adding the destination to itself will reset this operand. Thus copying an element requires two additions or four clock cycles. This is a relatively small price to pay (extra 4 clock cycles) considering that this operation is only performed once per point addition. Also, wide datapath (≈ 160 bits) would utilize a large amount of the routing resources if the feedback path was implemented.

Figure 9.4 shows the status of all memory locations in every state of the sequence. Since x_{k-1}, y_{k-1}, and z_{k-1} are needed in the computation stage, these memory locations (RAM0, RAM1, and RAM2) are marked as permanent in the figure. If a location is marked as permanent, no other data can be written into it for the rest of the sequence in a given stage. At the end of the precomputation sequence the memory bank is almost entirely filled with intermediate values that are passed to the calculation stage. After the thirteenth operation only three memory locations are unused. Consequently, the entire encryption engine can be realized with a minimum of thirteen memory elements.

#	*Operation*	*Result*	*Write*
1.	RAM6 = RAM6 + RAM6	reset RAM6	none
2.	RAM5 = RAM3 · RAM2	$x_0 z_{k-1}$	RAM6
3	RAM10 = RAM4 · RAM2	$y_0 z_{k-1}$	RAM5
4.	RAM5 = RAM0 + RAM5	$x_0 z_{k-1} + x_{k-1} = A$	RAM10
5.	RAM6 = RAM5 + RAM6	copy A to RAM6	RAM5
6.	RAM10 = RAM1 + RAM10	$y_0 z_{k-1} + y_{k-1} = B$	RAM6
7.	RAM11 = $(\text{RAM5})^2$	A^2	RAM10
8.	RAM6 = RAM10 + RAM6	$A + B = C$	RAM11
9.	RAM7 = RAM2 · RAM15	$a_2 z_{k-1}$	RAM6
10.	RAM8 = RAM2 · RAM10	$B z_{k-1}$	RAM7
11.	RAM7 = RAM5 + RAM7	$A + a_2 z_{k-1}$	RAM8
12.	RAM9 = RAM7 · RAM11	$A^2(A + a_2 z_{k-1})$	RAM7
13.	RAM7 = RAM8 · RAM6	$z_{k-1}BC$	RAM9
14.	RAM9 = RAM7 + RAM9	$A^2(A + a_2 z_{k-1}) + z_{k-1}BC = D$	RAM7

Table 9.3: Add_fsm precomputation sequence

Check:

$$\begin{aligned}
\text{RAM5} = \text{RAM0} + \text{RAM5} &= \text{RAM0} + \text{RAM3} \cdot \text{RAM2} \\
&= x_0 z_{k-1} + x_{k-1} = A \ \surd \\
\text{RAM10} = \text{RAM1} + \text{RAM10} &= \text{RAM1} + \text{RAM4} \cdot \text{RAM2} \\
&= y_0 z_{k-1} + y_{k-1} = B \ \surd \\
\text{RAM11} = (\text{RAM5})^2 &= A^2 \ \surd \\
\text{RAM6} = \text{RAM10} + \text{RAM6} &= A + B = C \ \surd \\
\text{RAM9} = \text{RAM7} + \text{RAM9} &= \text{RAM8} \cdot \text{RAM6} + \text{RAM7} \cdot \text{RAM11} \\
&= \text{RAM2} \cdot \text{RAM10} \cdot C + (\text{RAM5} + \text{RAM7})A^2 \\
&= \text{RAM2} \cdot B \cdot C + A^2(\text{RAM5} + \text{RAM2} \cdot \text{RAM15}) \\
&= z_{k-1}BC + A^2(z_{k-1}a_2 + A) = D \ \surd
\end{aligned}$$

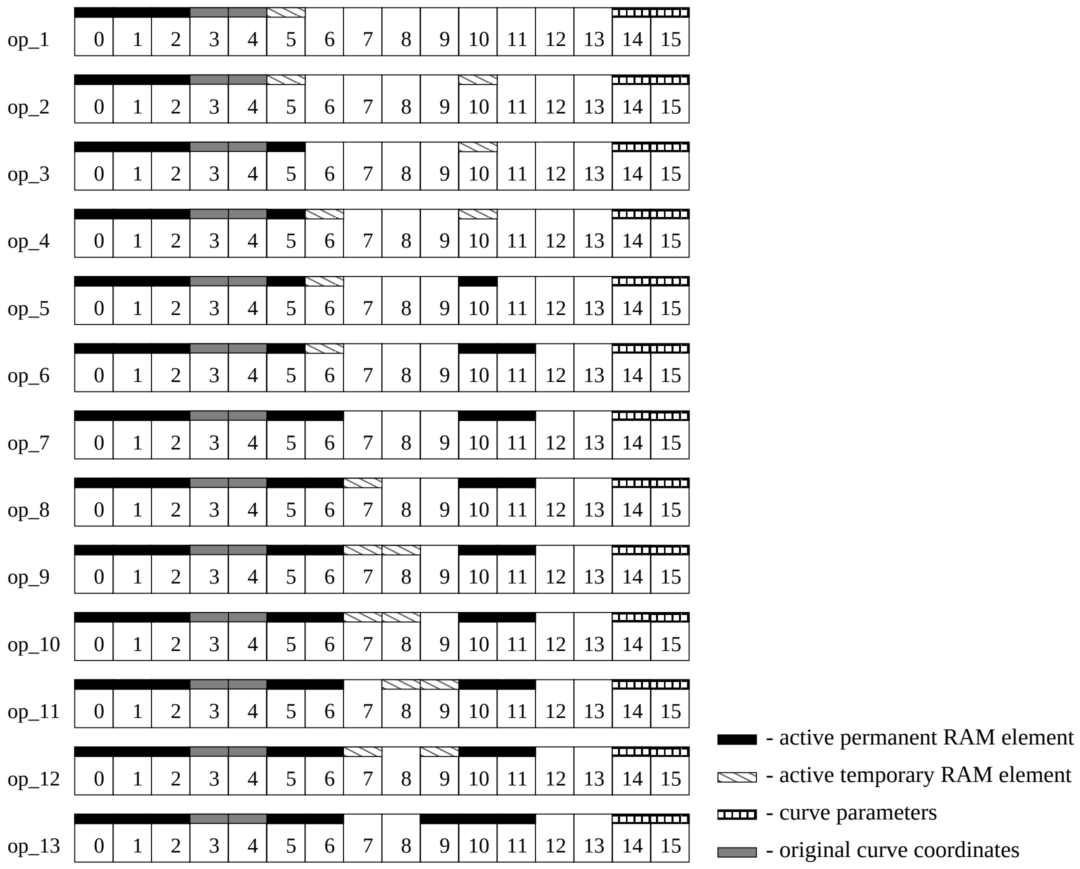

Figure 9.4: Memory usage for addition precomputation stage

Final State:

After the add precomputation stage memory bank contains the following results:

$$\begin{aligned}
&\text{RAM0} = x_{k-1}\\
&\text{RAM1} = y_{k-1}\\
&\text{RAM2} = z_{k-1}\\
&\text{RAM3} = x_0\\
&\text{RAM4} = y_0\\
&\text{RAM5} = A\\
&\text{RAM6} = C\\
&\text{RAM7} - \text{RAM8} = \text{empty}\\
&\text{RAM9} = D\\
&\text{RAM10} = B\\
&\text{RAM11} = A^2\\
&\text{RAM12} - \text{RAM13} = \text{empty}\\
&\text{RAM14} = a_6\\
&\text{RAM15} = a_2
\end{aligned}$$

At the end of this stage only four memory locations are empty. However the intermediate values in all other locations will be discarded immediately after they are used giving more storage area during the computation stage.

9.2.2 Computation Stage

Most of the work done to compute one point addition is done in the precomputation stage where all intermediate values are calculated. The computation stage uses these values to calculate x_k, y_k, and z_k thus the sequence in this stage is shorter than in the precomputation stage. The final result computes the following equations:

$$\begin{aligned}x_k &= AD\\ y_k &= CD + A^2(Bx_{k-1} + Ay_{k-1})\\ z_k &= A^3 z_{k-1}\end{aligned}$$

Initial State:

The initial values needed for this stage are as follows:

$$\begin{aligned}&\text{RAM0} = x_{k-1}\\ &\text{RAM1} = y_{k-1}\\ &\text{RAM2} = z_{k-1}\\ &\text{RAM3} = x_0\\ &\text{RAM4} = y_0\\ &\text{RAM5} = A\\ &\text{RAM6} = C\\ &\text{RAM7} - \text{RAM8} = \text{empty}\\ &\text{RAM9} = D\\ &\text{RAM10} = B\\ &\text{RAM11} = A^2\\ &\text{RAM12} - \text{RAM13} = \text{empty}\\ &\text{RAM14} = a_6\\ &\text{RAM15} = a_2\end{aligned}$$

Computation Sequence:

This sequence requires a maximum of twelve memory locations that are utilized in third operation (refer to Figure 9.5). It is important to note here that although only twelve locations are necessary in this stage, the precomputation stage uses thirteen

#	*Operation*	*Result*	*Write*
15.	RAM7 = RAM10 · RAM0	Bx_{k-1}	RAM9†
16.	RAM10 = RAM5 · RAM1	Ay_{k-1}	RAM7
17.	RAM0 = RAM5 · RAM9	$AD = x_k$	RAM10
18.	RAM8 = RAM6 · RAM9	CD	RAM0
19.	RAM7 = RAM10 + RAM7	$Ay_{k-1} + Bx_{k-1}$	RAM8
20.	RAM10 = RAM2 · RAM11	$A^2 z_{k-1}$	RAM7
21.	RAM1 = RAM7 · RAM11	$A^2(Ay_{k-1} + Bx_{k-1})$	RAM10
22.	RAM2 = RAM5 · RAM10	$A^3 z_{k-1} = z_k$	RAM1
23.	RAM1 = RAM8 + RAM1	$A^2(Ay_{k-1} + Bx_{k-1}) + CD = y_k$	RAM2
24.			RAM1

† - last result from precomputation stage

Table 9.4: Add_fsm state operations

which is the maximum number needed to implement the entire cryptosystem. Consequently, $n \cdot m \cdot 13$ bits of storage space is required, $n \cdot m \cdot 2$ of which is allocated for the curve parameters (a_2 and a_6).

Figure 9.5 shows memory allocation for the computation stage of point addition. At the end of the sequence, the result is available in the first three registers. This allows for a point doubling or point addition to be performed on the next clock cycle.

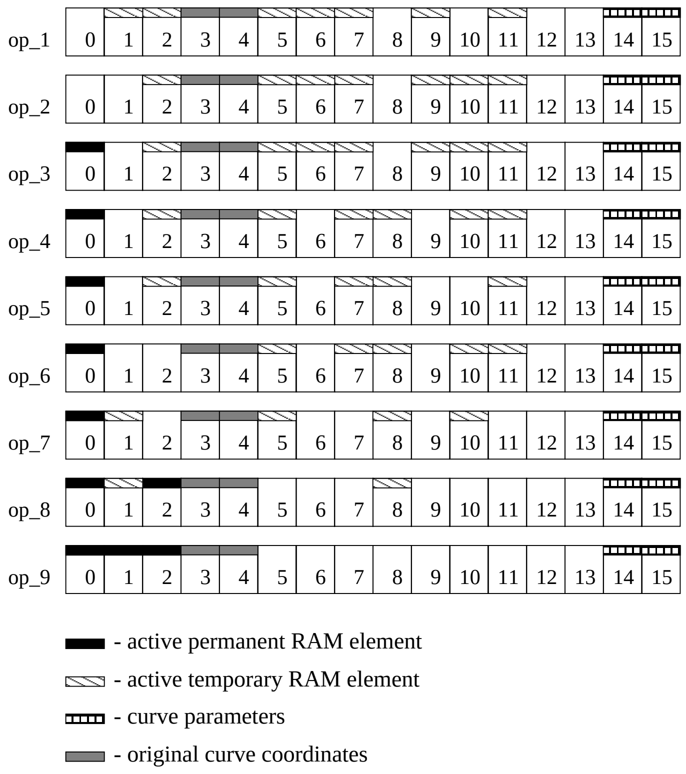

Figure 9.5: Memory usage for addition computation stage

Check:

$$
\begin{aligned}
\text{RAM0} = \text{RAM5} \cdot \text{RAM9} \quad &= AD = x_k \ \surd \\
\text{RAM1} = \text{RAM8} + \text{RAM1} \quad &= \text{RAM6} \cdot \text{RAM9} + \text{RAM7} \cdot \text{RAM11} \\
&= \text{RAM6} \cdot \text{RAM9} + (\text{RAM10} + \text{RAM7}) \cdot \text{RAM11} \\
&= \text{RAM6} \cdot \text{RAM9} + (\text{RAM5} \cdot \text{RAM1} + \text{RAM10} \cdot \text{RAM0}) \cdot \text{RAM11} \\
&= CD + (Bx_{k-1} + Ay_{k-1})A^2 = y_k \ \surd \\
\text{RAM2} = \text{RAM5} \cdot \text{RAM10} \quad &= \text{RAM5} \cdot \text{RAM2} \cdot \text{RAM11} \\
&= A \cdot z_{k-1} \cdot A^2 = z_{k-1}A^3 = z_k \ \surd
\end{aligned}
$$

Final State:

After this stage memory is returned to it's idle state. Mainly, the first the addresses are filled with the new coordinate $(x_k : y_k : z_k)$, the original coordinates, and the curve parameters being stored in the following location:

$$\begin{aligned}
&\mathrm{RAM0} = x_k \\
&\mathrm{RAM1} = y_k \\
&\mathrm{RAM2} = z_k \\
&\mathrm{RAM3} = x_0 \\
&\mathrm{RAM4} = y_0 \\
&\mathrm{RAM5} - \mathrm{RAM13} = \mathrm{empty} \\
&\mathrm{RAM14} = a_6 \\
&\mathrm{RAM15} = a_2
\end{aligned}$$

9.2.3 Summary

In this section, the results from the precomputation and computation stages are analyzed to generate formulae for storage requirements and maximum number of clock cycles necessary for point addition. First, a general case is presented followed by an example.

General Case

Storage Requirements: As previously mentioned, the precomputation stage requires 13 memory elements. This is the absolute maximum number of elements needed to realize the entire cryptosystem. For a secure elliptic curve system, $m \cdot n \approx 160$, thus a useful implementation will require $160 \cdot 13 = 2.08Kb$ of storage.

Clock Cycles: The following arithmetic operations are required in the precomputation stage:

$$\#MULT + \#SQUARE = 7$$
$$\#ADD = 7$$

Similarly, the computation stage requires:

$$\#MULT + \#SQUARE = 7$$
$$\#ADD = 2$$
$$\#WRITE = 1$$

Since multiplication/squaring requires $m + 1$ clock cycles, addition requires 2 clock cycles and last memory write is done in one clock cycle, the total number of clock cycles for the entire point addition is is

$$14 \cdot (m + 1) + 9 \cdot 2 + 1 = 14m + 33. \tag{9.2}$$

In this case m determines the number of slices of the composite field architecture. Comparing point add results with number of clock cycles necessary for point double operation, shows that point double can be computed in approximately 80% of the time required for point addition. This holds for $m = 21$.

Example GF$((2^8)^{21}) \equiv GF(2^{168})$

The example used in point doubling can be used to estimate point addition. In particular, GF$((2^8)^{21})$ is used.

Storage Requirements: A total of 13 memory elements are needed with each element having $8 \cdot 21 = 168$ bits. This results in total storage area of $2.184Kb$ of storage.

Clock Cycles: The total number of clock cycles required to add one point on this curve is $14 \cdot m + 33$ where $m = 21$. This results in a total of 327 clock cycles, each clock period being the length of time required for one subfield multiplication. Using the same clock estimate of 60 ns, the total time for one point addition is $60ns \cdot 327 = 19.62\mu s$.

9.3 State Machine Encoding

Both of the state machines that sequence group operations are implemented with two types of encoding schemes. In all, four state machines were developed: `Double_fsm_1hot`, `Add_fsm_1hot`, `Double_fsm_enum`, and `Add_fsm_enum`. The two one-hot state machines are used together to define the Group Operation Level control sequence. One-hot state machines are explicitly defined in VHDL so that only one bit is used for each state. This ensured that the synthesis tools did not apply a different encoding scheme to this control structure. The other two state machines are defined with the enumerated attribute. Coding VHDL for this type of state machines is much easier since the state vector is much shorter. Enumerated state machine includes an attribute that the synthesis tool understands and uses to develop optimal state encoding for a particular state machine. Thus by synthesizing the enumerated and one-hot state machines we can determine if one-hot encoding is appropriate for this design.

Chapter 10

Encryption Level

This chapter describes all control architectures associated with the Encryption Level. Here, we describe the implementation of the double-and-add algorithm and define the I/O interface.

10.1 Point Multiplication

The encryption level control sequence is responsible for all I/O operations as well as the necessary initialization of RAM locations. Point double and point add operations are controlled at the operation level described in Sections 9.1 and 9.2. However, point multiplication by an integer is an additional process that utilizes the operation level architecture. Depending on the length of the integer (often in the range of 160 bits) the double or add command has to be issued to the operation level state machine(s) which in turn calls the basic Galois field arithmetic functions. Thus the major task of this state machine is to perform the double-and-add algorithm according to the integer that is read from the input pins. In addition, the encryption level defines

State	*Operation*	*Clock Cycles*
IDLE	assert ready output signal	waits until START_JOB
LOAD	read point coordinates into memory	$5q$ †
DOUBLE	issue DOUBLE sequence command to DOUBLE_fsm	$12(m+2)$ ‡
ADD	issue ADD sequence command to ADD_fsm	$7(2m+5)$ ‡
UNLOAD	write result of calculated point to output pins	3 †

† - device/version specific definition

‡ - operation specific definition

Table 10.1: IO_fsm state operations

the input and output interface to the FPGA. The state machine (IO_fsm) is idle until the START_JOB signal is asserted by external logic. Besides the idle state, the IO_fsm can be in one of five states described in Table 10.1. The transitions between states are controlled by the clock, state counters, and the mult_vector that defines the integer used to perform point multiplication. The general description of the square-and-multiply algorithm is presented in [27] and can be adopted for the double-and-add method used in EC cryptosystem. Thus, the IO_fsm sequentially scans mult_vector one bit at a time starting with the second most significant bit. If a particular bit is "1", both double and add operations have to be performed. Only the double operation is issued when a bit in mult_vector is "0". Thus we have on average $mn - 1$ double commands and $(mn - 1) \div 2$ add commands. Figure 10.1 shows the structure of the IO_fsm with corresponding transitions. Depending on the size of the device and the available number of pins, the I/O control can be implemented to match the number of pins used to input the point coordinates. Our current implementation writes each coordinate into memory within three clock cycles (i.e., $q = 3$). In other words, only $(m \cdot n) \div 3$ pins are necessary to write the point coordinates. The LOAD

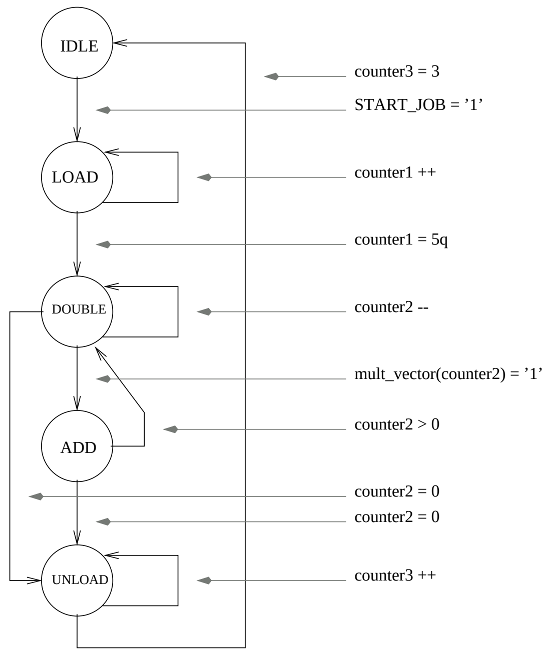

Figure 10.1: I/O finite state machine

Coordinate	*Operation*
x_0	load RAM0
	load RAM3
y_0	load RAM1
	load RAM4
z_0	load RAM2

Table 10.2: Load state of IO_fsm

state realizes five different operations described in Table 10.2 with each operation lasting q clock cycles. This results in a total of $5q$ clock cycles. During that time the entire multiplication vector can also be clocked into the flip-flops available in each IOB.

Once point multiplication is completed, the resulting coordinates which are stored in the top three memory locations, are introduced to the output pins. More specifically,

x_{new} is forwarded on the first clock cycle of the unload stage, y_{new} is forwarded on the second clock cycle, and z_{new} is forwarded on the last clock cycle of the unload stage. After the last coordinate is unloaded, the ready signal is asserted informing external logic that new computation can be started.

10.2 State Machine Encoding

This state machine that controls the encryption process was encoded using the enumerated type attribute. Thus, the task of choosing optimal state encoding scheme was left for the synthesis tools in this case. We found that the synthesis tool performed very well in defining the optimal state encoding scheme.

10.3 Point Multiplication Complexity

From the timing analysis performed in Sections 9.1.3 and 9.2.3, and the analysis of double-and-add algorithm described in this chapter, it is possible to derive the number of clock cycles necessary for the entire point multiplication in terms of the number of slices (m) and width of arithmetic functions (n). Since each double sequence requires $12m + 21$ (Equation 9.1) clock cycles and this operation is executed $mn - 1$ times, the total clock cycles allocated for all double operations is:

$$double_{total} = (12m + 21)(mn - 1) = 12m^2n + 21mn - 12m - 21$$

Similarly, each point addition requires $14m + 33$ (Equation 9.2) clock cycles and it is executed $(mn - 1) \div 2$ times. Thus the total clock cycles required for all point additions is:

$$add_{total} = \frac{(14m + 33)(mn - 1)}{2} = 7m^2n + 16.5mn - 7m - 16.5$$

The summation of both of these totals yields the number of clock cycles necessary for the entire point multiplication:

$$\#clk\_cyc = double_{total} + add_{total} = 19m^2n + 37.5mn - 19m - 37.5. \tag{10.1}$$

Equation 10.1 can be applied to our results presented in Chapter 11 to obtain the time required for entire point multiplication.

Chapter 11

Results And Timing Analysis

This chapter discusses the results gathered during the course of the thesis work. Section 11.1 gives a brief introduction outlining results from a system point of view. In Section 11.3 timing and area results are given for all major modules of the design. The effects of the arithmetic composition on area and timing are discussed. Finally, Section 11.4 derives timing and area performance of the entire system for each composition that was implemented.

11.1 Introduction

The final implementation was realized for an EC multiplications architecture for four different fields. Namely, $GF((2^4)^9)$, $GF((2^4)^{33})$, $GF((2^8)^{21})$, and $GF(((2^4)^2)^{21})$ arithmetic was used at the kernel of the architectures. These fields are isomorphic to $GF(2^{36})$, $GF(2^{132})$, $GF(2^{168})$, and $GF(2^{168})$ respectively. Obviously, the size of the final design depends on the finite field chosen. Because of the slice architecture of the design, implementing many different versions was possible with minor changes to the structure of the system. For example, changing the implementation from $GF((2^8)^{21})$ to $GF(((2^4)^2)^{21})$, where the subfield $GF(2^8)$ was replaced by the subfield $GF((2^4)^2)$ with bit-parallel arithmetic, only involved re-synthesis and re-compilation with different subfield arithmetic structures and input parameters defining the slice and width variables.

The device chosen for all implementations was the XC4062XLPG475-1. This device has enough CLBs to fit the largest of the developed architectures. In fact, this device has 2304 CLBs and the largest design used approximately 83% of the available area. As it turns out, the place and route tools posed the greatest challenge during the implementation phase of the design. The complexity of the design resulted in unacceptable compile times. The long compile times was the reason that a relatively small version over $GF((2^4)^9)$ was also constructed. As a matter of fact, only $GF((2^4)^9)$ could be completely placed and routed into a device. Moreover, the small design could only be implemented by relaxing timing constraints. This resulted in degraded performance as the minimum period was approximately 300ns.

11.2 Metric

The results presented in this thesis were gathered from two different tools. With the Synopsys tools which performed a design synthesis from the VHDL architecture description, we obtained estimated clock periods for all designs and all components of the design. In addition, area results were also reported by the Synopsys tools. In a few instances, the timing results were not reported by Synopsys. This is due to the use of LogiBlox components which are not interpreted by the Synopsys tools. Synthesis of components containing LogiBlox did not report timing results as that information is only known to the place and route tools. Utilization of LogiBlox does not greatly effect the Synopsys area reports because storage elements that were implemented with LogiBlox are placed in the same CLB as the combinatorial logic. As mentioned earlier, most flip-flops inside the CLBs are not utilized unless specific storage components are used. Even though Synopsys area reports only include utilization of combinatorial elements in the CLB (LUTs), we found that these results are quite accurate. This is apparent when we compare Synopsys to XILINX area results for a specific component. In most cases, area utilization reported by Synopsys is the same as the area reported by the place and route tools.

The placed and routed design was verified with the procedure described in Chapter 5. Point double, point addition and point multiplication were verified with back-annotated design. The currently available place and route tools experience exponential growth of compilation times as the complexity of the design increases. Since the small design ($GF((2^4)^9)$) took approximately 1.5 weeks (without any timing constraints), the larger designs would require a very long time. Consequently, the results presented in Section 11.4 show estimated area and clock speeds based synthesis results provided by `fpga_analyzer`. All of the designs are verified in RTL simulation. In addition, the smallest design verified with back-annotated simulation. Since the entire design is based on a slice architecture, verifying the correct operation of the smallest implementation indicates a very high likelihood that the remaining designs are correct also. The control mechanism functions are the same in all designs. The only difference is the exit condition from the multiplication state as the counter has to count to $(slice - 1)$ clocks.

The results presented here include CLB counts as well as minimum clock periods. The minimum clock period is a metric that defines the speed of the design. This measurement is a standard way in which system performance is measured. Area reports, on the other hand, provide a metric that is somewhat abstract when compared to gate equivalences. The XILINX CLB can be configured as logic gates and as memory. Depending on the implementation of the CLBs, gate count may vary. XILINX reports that depending on the design, a CLB can implement anywhere between 15 and 48 gates equivalence [46]. If the design does not contain any memory elements, the CLB is approximately equivalent to 15 CLBs. Additionally, a mixed design will yield up to 48 gates per CLB and the estimated typical number of gates per CLB is shown to be 28.5 [46]. We will use this metric to show the gate equivalence for system results.

11.3 Modules

This section describes results gathered for all individual components. Section 11.3.1 outlines results for combinatorial elements and Section 11.3.2 describes the clocked elements.

11.3.1 Combinatorial Components

In this section we compare results for all combinatorial components and outline the relationship between choice of finite field and area and timing results. As previously stated, we used the XILINX device XC4062XLPG475-1 to implement our design. The Xilinx tools provided information regarding maximum combinatorial delay and maximum path delay. Combinatorial delay is associated with the time required for all logic and net delays within the design. Maximum path delay describes the longest path within the design.

Addition Element

Table 11.1 shows the timing and area results for the addition module. As shown previously, the addition processing element is composed solely of XOR gates. Since addition in $GF(2^k)$ is a bitwise XOR function, all bits are computed in parallel resulting in very predictable timing performance. The only addition element that could be placed and routed was $GF((2^4)^9)$ because of limited number of input/output pads available on the XILINX devices. The device used for implementation includes 384 usable I/O pins and the larger designs require $3 \cdot 132$ and $3 \cdot 168$ I/O pins. The

		Synopsys Results		**Device Results**	
Module	*Composition*	*Area* (CLB)	*Path*	*Area* (CLB)	*Path*
`Add PE`	$GF((2^4)^9)$	36	8.48ns	36	8.51ns
	$GF((2^4)^{33})$	132	8.48ns	132 †	8.51ns †
	$GF((2^8)^{21})$	168	8.48ns	168 †	8.51ns †
	$GF(((2^4)^2)^{21})$	168	8.48ns	168 †	8.51ns †

† - extrapolated result

Table 11.1: Addition results

estimated results are very accurate as the structure of this component is very regular. Furthermore, the area results obtained from Synopsys can be generalized in terms of field order of the arithmetic composition. More specifically, the area utilization for addition element increases linearly with the field order. For an implementation in $GF((2^n)^m)$, $n \cdot m$ CLBs are required. In addition, because all bits are computed in parallel, the timing result for the smallest addition unit reflects the delay through the adder regardless of the width of the operation. Thus addition can be done in 8.48ns independent of the finite field chosen.

Switch 1

Switch 1 is used to route the correct data element to one of two processing elements. Because of the wide datapath, this component could only be placed and routed in its smallest form. Only $GF((2^4)^9)$ version was placed and routed and verified with back-annotated design. Consequently, the XILINX XC4062XLPG475-1 results for the remaining designs had to be estimated. This was done through a linear approximation. Table 11.2 shows area utilization and timing performance of this component. The Synopsys timing reports for $GF((2^4)^9)$ compare very well (a discrepancy of 2.23ns) with the results provided by the place and route tools. Once again, area utilization

		Synopsys Results		**Device Results**	
Module	*Composition*	*Area* (CLB)	*Path*	*Area* (CLB)	*Path*
Switch1	$GF((2^4)^9)$	35	35.40ns	36	37.63ns
	$GF((2^4)^{33})$	135	49.25ns	139 †	52.36ns †
	$GF((2^8)^{21})$	156	44.86ns	161 †	47.69ns †
	$GF(((2^4)^2)^{21})$	156	44.86ns	161 †	47.69ns †

† - extrapolated result

Table 11.2: Switch 1 results

scales linearly with the choice of the field. The added complexity associated with larger finite fields is marginally apparent from the timing results for this component. From these results, we can speculate that the combinatorial delay through Switch 1 is more dependent on the number of slices m defined in the architecture than on the actual field size. As it is apparent from Table 11.2, $GF((2^4)^{33})$ exhibits longer combinatorial delay than $GF((2^8)^{21})$ and $GF(((2^4)^2)^{21})$ which have larger field orders. This may be due to the physical constraint of the device. Since the die size is fixed, there is a limited number of rows and columns of CLBs in a particular device. Thus the architecture with fewer slices may be able to fit in one column.

Switch 2

Timing and area results for Switch 2 are shown in Table 11.3. Switch 2 is placed on the other side of the datapath routing results from the processing element to the RAM bank. This component also requires a wide datapath. Consequently, only $GF((2^4)^9)$ was verified after the place and route process. A linear area complexity with field choice is once again apparent. Additionally, synthesis results indicate that

		Synopsys Results		**Device Results**	
Module	*Composition*	*Area* (CLB)	*Path*	*Area* (CLB)	*Path*
Switch2	$GF((2^4)^9)$	37	22.08ns	37	27.88ns
	$GF((2^4)^{33})$	133	22.08ns	133 †	27.88ns †
	$GF((2^8)^{21})$	169	22.08ns	169 †	27.88ns †
	$GF(((2^4)^2)^{21})$	169	22.08ns	169 †	27.88ns †

† - extrapolated result

Table 11.3: Switch 2 results

combinatorial delay does not increase with the size of the field. This is because Switch 2 multiplexes all inputs in parallel. Comparing synthesis results with place and route results, we see that area results are identical and timing results have a discrepancy of 5.8ns.

Switch 3

Switch 3 is used to multiplex the control signals from Add_fsm and Double_fsm. This component is controlled by IO_fsm. Table 11.4 shows the synthesis and implementation results. Synthesis result for area utilization are very accurate. Since all compositions of Switch 3 could be placed and routed, it is possible to compare the timing results for the synthesized design with the implemented design. From Table 11.4, we can see that timing approximations provided by synthesis results compare very well

		Synopsys Results		**Device Results**	
Module	*Composition*	*Area* (CLB)	*Path*	*Area* (CLB)	*Path*
Switch3	$GF((2^4)^9)$	8	23.37ns	9	27.43ns
	$GF((2^4)^{33})$	8	23.37ns	8	21.52ns
	$GF((2^8)^{21})$	8	21.70ns	8	27.16ns
	$GF(((2^4)^2)^{21})$	8	21.70ns	8	25.00ns

Table 11.4: Switch 3 results

to the actual combinatorial delay obtained after the design is placed into a specific device.

Switch 5

Switch 5 is a large tri-state buffer. As shown in Table 11.5, this component does not use any CLBs since each tri-state gate is implemented in the I/O pad. The timing results provided by the XILINX tools is far greater than the expected results obtained through synthesis. This is due to the large path delay associated with routing the signal from the multiplexer to the pad. Since this component required many IO pads, the internal logic is placed so that all paths have similar delay. Consequently, the design is placed near the center of the device resulting in large path delays between the design and the pins. This of course is not an issue when the design is a component

		Synopsys Results		**Device Results**		
Module	*Composition*	*Area* (IOB)	*Path*	*Area* (IOB)	*Net*	*Path*
Switch5	$GF((2^4)^9)$	73	13.50ns	73	31.76ns	36.24ns
	$GF((2^4)^{33})$	265	13.50ns	265	55.86ns	60.34ns
	$GF((2^8)^{21})$	337	13.50ns	337	46.89ns	51.37ns
	$GF(((2^4)^2)^{21})$	337	13.50ns	337	46.89ns	51.37ns

Table 11.5: Switch 5 results

of the entire system. In fact, the maximum combinatorial delay becomes the delay through one CLB plus any delay associated with routing inputs to Switch 5 from other components in the system architecture.

11.3.2 Clocked Components

Synthesis and implementation of clocked components requires a careful selection of clock constraints. If this input parameter of the synthesis and place and route tools, is chosen too optimistically, long compilation times will result without significant increase in performance. Conversely, if the clock constraint is too relaxed, optimization will be limited. Consequently, synthesis and implementation of these components resulted in many iterations.

RAM Bank

As mentioned previously Synthesis results for the RAM bank are not available because only vendor specific components were used to implement this module. Thus, Table 11.6 lists the results provided by the place and route tools. The RAM bank is

Module	*Composition*	**Device Results**	
		Area (CLB)	*Timing* (clk_prd)
RAM Bank	$GF((2^4)^9)$	36	34.04ns
	$GF((2^4)^{33})$	132 †	34.04ns †
	$GF((2^8)^{21})$	168 †	34.04ns †
	$GF(((2^4)^2)^{21})$	168 †	34.04ns †

† - LogiBlox approximation

Table 11.6: RAM bank results

composed of slices of memory blocks that are all independent of each other. The same address and reset signals are routed to all RAM blocks. Due to its regular structure, the RAM bank clock period will have a fairly constant behavior for all finite fields that were implemented. Furthermore, CLB usage scales linearly with the size of the field.

Multiplication Element

The multiplication element also contains LogiBlox components inside the hybrid architecture. Besides the LogiBlox registers, however, this component implements combinatorial logic to realize arithmetic in the subfield. Consequently, the area reported by the synthesis tool is accurate but the timing cannot be derived. Table 11.7 shows

Module	*Composition*	**Synopsys Results**		**Device Results**	
		Area (CLB)	*Timing* (clk_prd)	*Area* (CLB)	*Timing* (clk_prd)
`Mult` PE	$GF((2^4)^9)$	73	NA	70	25.09ns
	$GF((2^4)^{33})$	265	NA	249	40.24ns
	$GF((2^8)^{21})$	673	NA	717	72.53ns
	$GF(((2^4)^2)^{21})$	778	NA	641	61.86ns

Table 11.7: Multiplication results

the results for the slice serial multiplier. The place and route results show that the internal structure of $GF((2^8)^{21})$ and $GF(((2^4)^2)^{21})$ plays an important role on the performance and area utilization. $GF(((2^4)^2)^{21})$ is smaller and faster than $GF((2^8)^{21})$ although both architectures have finite fields of the same size. This result verifies the findings in [37] bit parallel subfield multipliers were compared for FPGAs.

I/O Finite State Machine

Table 11.8 shows the results for the element controlling the encryption level algorithm. Area and timing results reported by synthesis and implementation tools are very close. It is interesting to point out that the minimum clock period for $GF((2^4)^{33})$

Module	*Composition*	**Synopsys Results**		**Device Results**	
		Area (CLB)	*Timing* (clk_prd)	*Area* (CLB)	*Timing* (clk_prd)
I/O FSM	$GF((2^4)^9)$	50	39.74ns	50	33.18ns
	$GF((2^4)^{33})$	95	68.59ns	97	69.06ns
	$GF((2^8)^{21})$	117	61.93ns	116	69.86ns
	$GF(((2^4)^2)^{21})$	117	61.93ns	116	69.86ns

Table 11.8: I/O state machine results

and $GF((2^8)^{21})$ is essentially identical. This is not surprising as the I/O state machine does not depend on the size of the finite field. The surprising result is that the smallest design $GF((2^4)^9)$ has a performance that is two times better than the remaining versions. We speculate that this is due to the few I/O buffers necessary for the small design in comparison to the larger versions.

Double Finite State Machine

Two versions of state machines controlling the Group Operation Level were implemented in the systems. One is one-hot encoded and the other was based on an attribute that is understood by the synthesis tools. More specifically, enumerated encoding was utilized. The use of enumerated encoding allows the synthesis tool to encode the state machine with the optimal solution. It is apparent from Table 11.9, that enumerated encoding style resulted in better area utilization for all architectures and increased performance for the larger designs, i.e., $GF((2^8)^{21})$ and $GF(((2^4)^2)^{21})$.

The one-hot encoded state machines were better suited for smaller field implemen-

Module	*Composition*	**Synopsys Results**		**Device Results**	
		Area (CLB)	*Timing* (clk_prd)	*Area* (CLB)	*Timing* (clk_prd)
Double_FSM (one hot)	$GF((2^4)^9)$	96	75.74ns	94	40.43ns
	$GF((2^4)^{33})$	99	75.98ns	100	40.90ns
	$GF((2^8)^{21})$	97	75.74ns	98	48.89ns
	$GF(((2^4)^2)^{21})$	97	75.74ns	98	48.89ns
Double_FSM (enum)	$GF((2^4)^9)$	67	51.60ns	68	35.18ns
	$GF((2^4)^{33})$	71	51.60ns	74	42.52ns
	$GF((2^8)^{21})$	69	51.04ns	71	44.61ns
	$GF(((2^4)^2)^{21})$	69	51.04ns	71	44.61ns

Table 11.9: Double state machine results

tation. In addition, both area and timing results do not change drastically when a different finite field is considered for the enumerated state machine. This is because the complexity of the state machines does not increase significantly with field order. For the one-hot encoded state machine, synthesis timing reports are much higher than the actual implementation. This shows that the synthesis tools performed better with enumerated encoding and that the XILINX tools had to perform more optimizations with one-hot encoded state machine. This was true in both cases, Double_fsm and Add_fsm.

Add Finite State Machine

The addition state machine is larger than the double state machine as described by the sequence in Chapter 9. Consequently area utilization is increased and performance is degraded. The results presented in Table 11.10 show similar behavior to the results obtained for the double state machine. This is because the general structure of both state machines is the same but the add state machine realizes a longer sequence.

Module	*Composition*	**Synopsys Results**		**Device Results**	
		Area (CLB)	*Timing* (clk_prd)	*Area* (CLB)	*Timing* (clk_prd)
Add_FSM (one hot)	$GF((2^4)^9)$	125	117.10ns	126	62.94ns
	$GF((2^4)^{33})$	130	120.23ns	133	64.28ns
	$GF((2^8)^{21})$	127	116.18ns	132	61.77ns
	$GF(((2^4)^2)^{21})$	127	116.18ns	132	61.77ns
Add_FSM (enum)	$GF((2^4)^9)$	89	51.36ns	91	45.92ns
	$GF((2^4)^{33})$	103	53.52ns	106	55.63ns
	$GF((2^8)^{21})$	85	55.01ns	89	48.09ns
	$GF(((2^4)^2)^{21})$	85	55.01ns	89	48.09ns

Table 11.10: Add state machine results

11.4 System

Synthesis and implementation of individual components shows the relative timing and area results for each element as a stand-alone unit. However, once these components are placed into the system, optimization can be performed between boundaries. This results in better area utilization as modules can be packed together and optimized to make full use of a CLB. This can be shown by summing up all area place and route results for $GF((2^4)^9)$:

$$Total_{area} = 126 + 94 + 50 + 70 + 36 + 73 + 9 + 37 + 36 + 36 = 567.$$

This number, however, is by 6% larger than the 535 CLBs used for the complete design. The summation yields a higher count than the actual implementation as depicted in Table 11.11. This shows that boundary optimization can reduce the number of CLBs used to implement the system.

Table 11.11 also shows the estimated number of gate equivalences for each architecture. The largest design utilizes $\approx 54,000$ gates which could be mapped to a

			Synopsys Results		
Module	*Composition*	$n \cdot m$	*Area* (CLB)	*Gate Equiv.* (est.)	*Timing* (clk_prd)
System (one hot)	$GF((2^4)^9)$	36	520	14820	78.4ns
	$GF((2^4)^{33})$	132	1249	35597	89.5ns
	$GF((2^8)^{21})$	168	1870	53295	88.9ns
	$GF(((2^4)^2)^{21})$	168	1891	53894	88.9ns
System (enum)	$GF((2^4)^9)$	36	577	16445	49.5ns
	$GF((2^4)^{33})$	132	1199	34172	68.5ns
	$GF((2^8)^{21})$	168	1810	51585	61.3ns
	$GF(((2^4)^2)^{21})$	168	1894	53979	61.3ns

Table 11.11: Point multiplication after synthesis

very small ASIC device. All architectures result in a clock period less than 100ns. Table 11.11 only shows results obtained through synthesis (with the exception of $GF((2^4)^9)$). As mentioned previously, implementation results cannot be obtained due to the long compile times of the place and route tools. However, the data presented in Section 11.3 shows that synthesis reports are, for the most part, very accurate when compared with the final device implementation. Comparing area results for $GF((2^4)^9)$, we can see that the results differ by only 3%. This leads us to speculate that such results are achievable and can be implemented with better place and route tools. Furthermore, in an commercial environment, there is a greater emphasis on implementation resulting in availability of better tools and supporting hardware that can tackle the job of placing our design into the XILINX device. We were able to verify our design in back-annotated form, only by relaxing the clock, resulting in decreased performance. However, synthesis results have been shown to be quite accurate, thus by using these values obtained from Synopsys, we were able to estimate the time required for one point multiplication.

Using Equation 10.1 from Section 10.3, and the synthesis results presented in Table 11.11, it is possible to estimate absolute timings for one point multiplication.

These results are presented in Table 11.12. According to synthesis results, multipli-

Module	*Composition*	$n \cdot m$	*# of cycles*	*clk_prd*	*Pt. Mult.*
System (one hot)	$GF((2^4)^9)$	36	7297.5	78.4ns	0.57ms
	$GF((2^4)^{33})$	132	87049.5	89.5ns	7.79ms
	$GF((2^8)^{21})$	168	72895.5	88.9ns	6.49ms
	$GF(((2^4)^2)^{21})$	168	72895.5	88.9ns	6.49ms
System (enum)	$GF((2^4)^9)$	36	7297.5	49.5ns	0.36ms
	$GF((2^4)^{33})$	132	87049.5	68.5ns	5.97ms
	$GF((2^8)^{21})$	168	72895.5	61.3ns	4.47ms
	$GF(((2^4)^2)^{21})$	168	72895.5	61.3ns	4.47ms

Table 11.12: Point multiplication results

cation of a point on the curve with arithmetic in $GF((2^8)^{21})$ or $GF(((2^4)^2)^{21})$ can be done in 4.47 milliseconds. Thus the two implementations with field order of 168 have a data throughput of 37.583kb/sec. This results is better than the best software implementation (7.8ms) on a 165MHz DEC Alpha [39], using the somewhat smaller field $GF(2^{155})$. All of these results are based on synthesis estimations of the minimum clock period and area utilization.

Chapter 12

Summary and Recommendations

12.1 Summary

From a design point of view, FPGAs provide a suitable environment for our implementation. These register rich devices can accommodate large memory structures and provide optimized macro cells that improve the speed performance of the system. The fine grain device architecture allows for synthesis tools to perform optimization almost at a gate level resulting in very efficient implementations.

The concept of reconfigurable hardware for elliptic curves is very attractive for various reasons. Reconfigurable hardware provides a versatile environment that is desirable when implementing modern cryptographic protocols. In the work described here, we have shown that an elliptic curve cryptosystem can principally be implemented on reconfigurable devices. There is however one limitation. The long compile times required to place and route the EC design into a specific device are currently a bottleneck during the development cycle. The available tools are improving very rapidly and new, larger devices are being offered from many vendors every year.

These improvements will make it possible to implement large and very complicated designs in the near future.

With the synthesis tools available, it was possible to obtain estimated results for all architectures. Furthermore, comparison of synthesis and implementation results, for various large modules of our design, shows that synthesis results are very accurate. Thus EC crypto engine can be implemented on XILINX FPGAs at the estimated computation time of approximately 4.5msec.

12.2 Recommendations for Future Research

This thesis concentrated on achieving point multiplication on elliptic curves in reconfigurable hardware. To our knowledge, this approach has not been yet attempted before. Below, we summarize some of the more important work that could still be done from a design and implementation point of view.

12.2.1 New Design Considerations

We would recommend to investigate different alternatives for implementing the control structure. For example, the possibility of using RAM and counters to generate the control vectors could be implemented.

Also, we would have liked to implement the system using two clocks to speed up computation times.

Another important design alternative that should be researched further is the implementation of multiple arithmetic processing elements. This would allow for parallel operation effectively reducing the entire computation cycle by half. Such an alternative would also require more routing resources.

Conversely, we would like to implement another design with a narrower datapath. Reducing datapath would result in longer computation cycle. However, such a design would allow us to use smaller FPGAs and possibly implement the general design on future smart cards.

12.2.2 Implementation Alternatives

From an implementation point of view, further research can be done to investigate other reconfigurable devices. Soft macros can be re-mapped so that the design can be implemented in EPLDs and CPLDs. Furthermore, devices from other vendors like ALTERA, AT&T and Motorola could be used to implement our design. This would allow us to research other place and route tools that may or may not perform better.

Future work could also concentrate on the actual system hardware implementation. For instance, designing a PC plug-in board with reconfigurable cryptographic algorithms seems like an attractive application.

Lastly, we would like to devote some time to try out one of the new devices that will be available from XILINX in the near future. The new Virtex family of devices use 0.25 micron, five layer metal process technology which will increase area, routing resources, and speed performance.

12.3 Concluding Remarks

In summary, we are very hopeful that the work presented here will provide some insight into hardware implementation of complex cryptographic algorithms. Point multiplication on elliptic curves is one of the most challenging computations used to implement public-key protocols. This holds especially true for hardware implementations of which very few have been reported in the literature. It is our intention to provide the reader with the issues concerning hardware implementation of elliptic curves. Moreover, one of our goals was to show that cryptographic protocols can be implemented in reconfigurable hardware. Wide datapaths associated with elliptic curve implementation in hardware is of concern when trying to use FPGA devices. However the limitation lies more in the tools rather than the resources available to us.

In this thesis, we have shown that reconfigurable hardware is a viable solution for public-key cryptography. In principal, elliptic curve point multiplication can be achieved on FPGAs resulting in very flexible implementation with increased speed performance over current software solution. As security issues become more and more pronounced in the next few years and supporting FPGA tools improve, we hope that reconfigurable hardware and elliptic curves will provide a viable solution.

Appendix A

Test Bench Sample

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_textio.all;
use work.array_types.all;

--
-- VHDL Architecture ALU.ALUtester.flow
--
-- Created:
--          by - mrosner.mrosner (pike.WPI.EDU)
--          at - 16:45:58 09/15/97
--
entity SYSTEM_TESTBENCH is
end SYSTEM_TESTBENCH;


ARCHITECTURE TEST OF SYSTEM_TESTBENCH IS

-- Architecture declarations
    CONSTANT clk_prd : time := 50 ns;
    constant width            : positive := 8;  --< bits in signle register
    constant slices           : positive := 21; --< number of registers in array
    constant depth            : positive := 4;
    constant trinomial_coeff  : positive := 2;  --< (k) from p. 158 in Menezes
    constant block_size       : positive := 7;

    SIGNAL clk             : std_logic ;
    SIGNAL reset           : std_logic ;
    SIGNAL START_JOB : std_logic;
    SIGNAL input_coord_pin  : input_slice_array;
    SIGNAL mult_vector : std_logic_vector((slices*width)-1 DOWNTO 0);

    SIGNAL ram_out          : slice_array;
    SIGNAL ready  : std_logic ;

    SIGNAL iclk : std_logic;

        PROCEDURE wait_clock(CONSTANT clk_ticks:integer) IS
        VARIABLE i : integer := 0;
        BEGIN
                FOR i IN 1 TO clk_ticks*2 LOOP
                        WAIT UNTIL iclk'EVENT;
                END LOOP;
        END wait_clock;

        component system
            PORT(
                 clk             : IN std_logic ;
                 reset           : IN std_logic ;
                 START_JOB      : IN std_logic;
                 input_coord_pin : IN input_slice_array;
                 mult_vector : IN std_logic_vector((slices*width)-1 DOWNTO 0);

                 ram_out     : OUT slice_array;
                 ready       : OUT std_logic
            );
        end component;

BEGIN
        UUT : system
                Port Map (
                          clk => clk,
                          reset =>reset,
                          START_JOB =>START_JOB,
                          input_coord_pin => input_coord_pin,
                          mult_vector =>mult_vector,

                          ram_out =>ram_out,
                          ready =>ready
                         );

   flow_process: PROCESS

   -- Process declarations
   VARIABLE j          : integer := 0;
   VARIABLE d_var      : unsigned(9 DOWNTO 0) := "0000000000";
   VARIABLE b_var      : unsigned(3 DOWNTO 0) := "0000";
   BEGIN
      input_coord_pin(0) <= "00110011";
      input_coord_pin(1) <= "00110011";
      input_coord_pin(2) <= "00110011";
      input_coord_pin(3) <= "00110011";
      input_coord_pin(4) <= "00110011";
      input_coord_pin(5) <= "00110011";
      input_coord_pin(6) <= "00110011";


      mult_vector <=   "1011011010110110101101101011011010110110
1011011010110110101101101011011010110110
1011011010110110101101101011011010110110
101101101011011010110110101101101011011010110110";

      reset <= '1';
      wait_clock(1);
      reset <= '0';
      START_JOB <= '1';
      wait_clock(1);
      START_JOB <= '0';
      wait_clock(6);
      input_coord_pin(0) <= "10011001";
      input_coord_pin(1) <= "10011001";
      input_coord_pin(2) <= "10011001";
      input_coord_pin(3) <= "10011001";
      input_coord_pin(4) <= "10011001";
      input_coord_pin(5) <= "10011001";
      input_coord_pin(6) <= "10011001";

      wait_clock(6);
      input_coord_pin(0) <= "01110111";
      input_coord_pin(1) <= "01110111";
      input_coord_pin(2) <= "01110111";
      input_coord_pin(3) <= "01110111";
      input_coord_pin(4) <= "01110111";
      input_coord_pin(5) <= "01110111";
      input_coord_pin(6) <= "01110111";
      wait_clock(3);

      wait;
   END PROCESS flow_process;

        -- Architecture concurrent statements
        clock_gen : PROCESS
        BEGIN
                iclk <= '0';
                WAIT FOR clk_prd/2;
                iclk <= '1';
                WAIT FOR clk_prd/2;
        END PROCESS clock_gen;
   clk <= iclk;

END TEST;

configuration CFG_TB of SYSTEM_TESTBENCH is
        for TEST
                for UUT : system
                        use configuration WORK.CFG_SYSTEM_RTL;
                end for;
        end for;
end CFG_TB;
```

Appendix B

C Model

```
/*******************************************************************
  This program is an Elliptic Curve emulator used to test a harware
  implementation developed for FPGAs. This is in partial fullfillment
  of a MS Thesis Degree at Worcester Polytechnic Intitute.

  The current version of the program uses the smalles subfield multiplier
  of GF(2^4) from which the architecture for GF(2^4)^2 is developed. This
  arithmetic unit is then used to build the hybrid multiplier (hyb_mult)
  and the hybrid adder (hyb_add). The hybrid adder performs a bitwise
  XOR of two operand vectors.

  Hyb_mult and hyb_add are then used to build on point addition
  module (pt_add) and one point multiplication module (pt_mult).

  Finaly, pt_add and pt_mult are used in conjunction with the
  square-and-multiply algorithm to realize a multiplication
  of a point on the curve bu an integer.

  Input files:  data_inX, data_inY, data_inZ -> X,Y,Z coord of original point
                curve_def1 -> elliptic curve parameter (a_2)
                curve_def2 -> elliptic curve parameter (a_6)
                mult_in -> 168 bit multiplier used in S-A-M
  programmer: Martin Rosner
  last update: February 12, 1998
 ******************************************************************/

#include <stdio.h>
#include "gfopsn.h"
#define SLICE 21
#define FEEDBACK 2
#define SLICE_WIDTH 8
```

```
typedef struct cmp_bus_tag{
  gfelt bus_lower;
  gfelt bus_higher;
}cmp_bus;


void multcomp(cmp_bus *a_in,
cmp_bus *b_in,
cmp_bus *res_out);

void sumcomp(cmp_bus *a_in,
cmp_bus *b_in,
cmp_bus *res_out);

void copy_vector(cmp_bus original[SLICE],
cmp_bus new_copy[SLICE]);

void hyb_mult(cmp_bus u_in[SLICE],
cmp_bus v_in[SLICE],
cmp_bus w_out[SLICE]);

void hyb_add(cmp_bus u_in[SLICE],
cmp_bus v_in[SLICE],
cmp_bus w_out[SLICE]);
void reset_result(cmp_bus result[SLICE]);

void pt_double(cmp_bus a_opX[SLICE],
cmp_bus a_opY[SLICE],
cmp_bus a_opZ[SLICE],
        cmp_bus curve_def1[SLICE],
        cmp_bus c_opX[SLICE],
cmp_bus c_opY[SLICE],
cmp_bus c_opZ[SLICE]);

void pt_add(cmp_bus a_opX[SLICE], cmp_bus a_opY[SLICE],
cmp_bus a_opZ[SLICE], cmp_bus b_opX[SLICE],
cmp_bus b_opY[SLICE], cmp_bus b_opZ[SLICE],
        cmp_bus curve_def2[SLICE], cmp_bus c_opX[SLICE],
cmp_bus c_opY[SLICE], cmp_bus c_opZ[SLICE]);

void print_vector(cmp_bus vector[SLICE]);

void scan_point(cmp_bus a_inX[SLICE], cmp_bus a_inY[SLICE],
cmp_bus a_inZ[SLICE], cmp_bus b_inX[SLICE],
cmp_bus b_inY[SLICE], cmp_bus b_inZ[SLICE],
                cmp_bus curve_def1[SLICE], cmp_bus curve_def2[SLICE]);
```

```
void scan_multiplier(int multiplier[SLICE]);

void  curve_mult(cmp_bus a_inX[SLICE], cmp_bus a_inY[SLICE],
cmp_bus a_inZ[SLICE], cmp_bus curve_def1[SLICE],
cmp_bus curve_def2[SLICE], cmp_bus res_outX[SLICE],
cmp_bus res_outY[SLICE], cmp_bus res_outZ[SLICE],
              int multiplier[SLICE]);

void scan_array(int multiplier[SLICE], int *outer_index, int *inner_index);

/******************************************************************/
/******************************************************************/
void main(void)
{
  cmp_bus a_inX[SLICE];
  cmp_bus a_inY[SLICE];
  cmp_bus a_inZ[SLICE];
  cmp_bus b_inX[SLICE];
  cmp_bus b_inY[SLICE];
  cmp_bus b_inZ[SLICE];
  cmp_bus res_outX[SLICE];
  cmp_bus res_outY[SLICE];
  cmp_bus res_outZ[SLICE];
  cmp_bus curve_def1[SLICE];
  cmp_bus curve_def2[SLICE];
  int multiplier[SLICE];

  reset_result(res_outX);
  reset_result(res_outY);
  reset_result(res_outZ);


  scan_multiplier(multiplier);
  scan_point(a_inX, a_inY, a_inZ,
             b_inX, b_inY, b_inZ,
             curve_def1, curve_def2);

  printf("\n*********\nOriginal point coordinates:\n");
  printf("X-COORD\n");
  print_vector(a_inX);
  printf("\nY-COORD\n");
  print_vector(a_inY);
  printf("\nZ-COORD\n");
  print_vector(a_inZ);

  printf("\n*********Computing new point...\n");
  curve_mult(a_inX, a_inY, a_inZ,
             curve_def1, curve_def2,
             res_outX, res_outY, res_outZ,
             multiplier);

  printf("DONE...\n");

  printf("\n*********\nNew point coordinates:\n");
  printf("X-COORD\n");
  print_vector(res_outX);

  printf("\nY-COORD\n");
  print_vector(res_outY);

  printf("\nZ-COORD\n");
  print_vector(res_outZ);
  printf("\n");
}


/******************************************************************/
/* the following compute one product element in the Galois field  */
/******************************************************************/
void multcomp(cmp_bus *a_in, cmp_bus *b_in, cmp_bus *res_out)
{
  gfelt cnst;
/*intermediate results*/
  gfelt cmult1, cmult2, cmult3, cadd1, cadd2, conout;

/*subfield polynomial modulus */
  f = "10011";
/*subfield length */
  N = 4;
  gfinit();

/*constant multiplier*/
  cnst = primpower(14);
  cmult1 = prod(a_in->bus_lower, b_in->bus_lower);
  cadd1 = sum(a_in->bus_lower, a_in->bus_higher);
  cadd2 = sum(b_in->bus_lower, b_in->bus_higher);
  cmult2 = prod(cadd1, cadd2);
  cmult3 = prod(a_in->bus_higher, b_in->bus_higher);
  conout = prod(cmult3, cnst);
  res_out->bus_higher = sum(cmult1, cmult2);
  res_out->bus_lower = sum(cmult1, conout);
}


/******************************************************************/
/* the following compute one sum element in the Galois field  */
/******************************************************************/
void sumcomp(cmp_bus *a_in, cmp_bus *b_in, cmp_bus *res_out)
```

```
{
  f = "10011"; /*subfield polynomial modulus*/
  N = 4; /*subfield length*/
  gfinit();
  res_out->bus_higher = sum(a_in->bus_higher, b_in->bus_higher);
  res_out->bus_lower = sum(a_in->bus_lower, b_in->bus_lower);
}

void hyb_add(cmp_bus u_in[SLICE], cmp_bus v_in[SLICE], cmp_bus w_out[SLICE])
{
  int i;
  int slices = SLICE;

  for(i=0; i<slices; i++){
    sumcomp(&u_in[i],
            &v_in[i],
            &w_out[i]);
  }
}

/*******************************************************************/
/*******************************************************************/
void hyb_mult(cmp_bus u_in[SLICE], cmp_bus v_in[SLICE], cmp_bus w_out[SLICE])
{
  int slices = SLICE;
  int feed_coef = FEEDBACK;
  int i,j;

  cmp_bus prod1[SLICE];
  cmp_bus clk_res[SLICE];
  cmp_bus temp;

  reset_result(w_out);
  for(i=slices-1; i>-1; i--){
    for(j=0; j<slices; j++){

      multcomp(&u_in[i],
               &v_in[j],
               &prod1[j]);

      if(j==0){
        /*first slice*/
        sumcomp(&prod1[j],
                &w_out[slices-1],
                &clk_res[j]);
      }
      else if(j==feed_coef){
        /*feedback slice*/
        sumcomp(&prod1[j],
                &w_out[j-1],
                &temp);
        sumcomp(&temp,
                &w_out[slices-1],
                &clk_res[j]);
      }
      else {
        /*all other slices*/
        sumcomp(&prod1[j],
                &w_out[j-1],
                &clk_res[j]);
      }
    }
      copy_vector(clk_res, w_out);
  }
}

/*****************************************************************/
/*****************************************************************/
void copy_vector(cmp_bus original[SLICE], cmp_bus new_copy[SLICE])
{
  int i;
  int slices = SLICE;

  for(i=0; i<slices; i++){
    new_copy[i].bus_higher = original[i].bus_higher;
    new_copy[i].bus_lower = original[i].bus_lower;
  }
}

void reset_result(cmp_bus result[SLICE])
{
  int i;
  int slices = SLICE;

  for(i=0; i<slices; i++){
    result[i].bus_higher = 0x0;
    result[i].bus_lower = 0x0;
  }
}


/*****************************************************************/
/*****************************************************************/
void pt_double(cmp_bus a_opX[SLICE], cmp_bus a_opY[SLICE],
cmp_bus a_opZ[SLICE], cmp_bus curve_def1[SLICE],
                cmp_bus c_opX[SLICE], cmp_bus c_opY[SLICE],
cmp_bus c_opZ[SLICE])
{
  cmp_bus tmp1[SLICE]; /*A*/
  cmp_bus tmp2[SLICE];
  cmp_bus tmp3[SLICE];
  cmp_bus tmp4[SLICE];
  cmp_bus tmp5[SLICE];
  cmp_bus tmp6[SLICE];
  cmp_bus tmp7[SLICE];

  /*begin precomp stage*/

  hyb_mult(a_opZ, a_opZ, tmp4);     /*spd1*/

  hyb_mult(a_opX, a_opX, tmp5);     /*sdp2*/

  hyb_mult(tmp4, tmp4, tmp6);       /*spd3*/

  hyb_mult(tmp5, tmp5, tmp4);       /*spd4*/

  hyb_mult(tmp6, curve_def1, tmp7); /*mpd1*/

  hyb_mult(a_opX, a_opZ, tmp3);     /*mpd2*/

  hyb_add(tmp4, tmp7, tmp1);
  copy_vector(tmp1, tmp7);          /*apd1*/
  /*end of precomp stage*/

  /*begin computation stage*/
  hyb_mult(a_opY, a_opZ, tmp6);     /*mcd1*/

  hyb_add(tmp3, tmp5, tmp1);
  copy_vector(tmp1, tmp5);          /*acd1*/

  hyb_mult(tmp3, tmp3, tmp2);       /*scd1*/

  hyb_add(tmp5, tmp6, tmp1);
  copy_vector(tmp1, tmp6);          /*acd2*/

  hyb_mult(tmp2, tmp3, c_opZ);      /*mcd2*/

  hyb_mult(tmp6, tmp7, tmp2);       /*mcd3*/

  hyb_mult(tmp4, tmp3, tmp6);       /*mcd4*/

  hyb_mult(tmp7, tmp3, c_opX);      /*mcd5*/

  hyb_add(tmp6, tmp2, c_opY);       /*acd3*/

  /*end of computation stage*/
}
```

```
/******************************************************************/
/******************************************************************/
void pt_add(cmp_bus a_opX[SLICE], cmp_bus a_opY[SLICE],
cmp_bus a_opZ[SLICE], cmp_bus b_opX[SLICE],
cmp_bus b_opY[SLICE], cmp_bus b_opZ[SLICE],
             cmp_bus curve_def2[SLICE], cmp_bus c_opX[SLICE],
cmp_bus c_opY[SLICE], cmp_bus c_opZ[SLICE])
{

  cmp_bus tmp1[SLICE];
  cmp_bus tmp2[SLICE];
  cmp_bus tmp3[SLICE];
  cmp_bus tmp4[SLICE];
  cmp_bus tmp5[SLICE];
  cmp_bus tmp6[SLICE];
  cmp_bus tmp7[SLICE];
  cmp_bus tmp8[SLICE];
  cmp_bus tmp9[SLICE];

  hyb_mult(b_opX, a_opZ, tmp5);      /*mpa1*/

  hyb_mult(b_opY, a_opZ, tmp3);      /*mpa2*/

  hyb_add(tmp5, a_opX, tmp1);        /*apa1*/
  copy_vector(tmp1, tmp5);

  copy_vector(tmp5, tmp6);           /*apa2*/

  hyb_add(tmp3, a_opY, tmp1);        /*apa3*/
  copy_vector(tmp1, tmp3);

  hyb_mult(tmp5, tmp5, tmp4);        /*spa1*/

  hyb_add(tmp3, tmp6, tmp1);         /*apa4*/
  copy_vector(tmp1, tmp6);

  hyb_mult(a_opZ, curve_def2, tmp7); /*mpa3*/

  hyb_mult(a_opZ, tmp3, tmp8);       /*mpa4*/

  hyb_add(tmp5, tmp7, tmp1);         /*apa5*/
  copy_vector(tmp1, tmp7);

  hyb_mult(tmp7, tmp4, tmp9);        /*mpa5*/

  hyb_mult(tmp8, tmp6, tmp7);        /*mpa6*/

  hyb_add(tmp7, tmp9, tmp1);         /*apa6*/
  copy_vector(tmp1, tmp9);

  hyb_mult(tmp3, a_opX, tmp7);       /*mca1*/

  hyb_mult(tmp5, a_opY, tmp3);       /*mca2*/

  hyb_mult(tmp5, tmp9, c_opX);       /*mca3*/

  hyb_mult(tmp6, tmp9, tmp8);        /*mca4*/

  hyb_add(tmp3, tmp7, tmp1);         /*aca1*/
  copy_vector(tmp1, tmp7);

  hyb_mult(tmp4, a_opZ, tmp3);       /*mca5*/

  hyb_mult(tmp4, tmp7, tmp2);        /*mca6*/

  hyb_mult(tmp3, tmp5, c_opZ);       /*mca7*/

  hyb_add(tmp8, tmp2, c_opY);        /*aca2*/
}

/*****************************************************************/
/*****************************************************************/
void scan_point(cmp_bus a_inX[SLICE], cmp_bus a_inY[SLICE],
cmp_bus a_inZ[SLICE], cmp_bus b_inX[SLICE],
cmp_bus b_inY[SLICE], cmp_bus b_inZ[SLICE],
                cmp_bus curve_def1[SLICE], cmp_bus curve_def2[SLICE])
{
  int index;
  FILE *inp1;
  FILE *inp2;
  FILE *inp3;
  FILE *inp4;
  FILE *inp5;
  FILE *outp1;

  index = 0;
  if((inp1 = fopen("data_inX", "r")) == NULL)
    {
      printf("Open Failed\n");
      /*end(1);*/
    }
  if((inp2 = fopen("data_inY", "r")) == NULL)
    {
      printf("Open Failed\n");
      /*end(1);*/
    }
  if((inp3 = fopen("data_inZ", "r")) == NULL)
```

```
    {
      printf("Open Failed\n");
      /*end(1);*/
    }
  if((inp4 = fopen("curve_def1", "r")) == NULL)
    {
      printf("Open Failed\n");
      /*end(1);*/
    }
  if((inp5 = fopen("curve_def2", "r")) == NULL)
    {
      printf("Open Failed\n");
      /*end(1);*/
    }
  if((outp1 = fopen("data_out1", "w+")) == NULL)
    {
      printf("Open Failed\n");
      /*end(1);*/
    }
  printf("\nnow scanning x-coord ...\n");
  while((fscanf(inp1, "%x %x %x %x\n",
                &(a_inX[index]).bus_higher,
&(a_inX[index]).bus_lower,
                &(b_inX[index]).bus_higher,
&(b_inX[index]).bus_lower))==4){

    printf("...slice %d: 1st pt -> %x%x; 2nd point -> %x%x\n",
           index,
           a_inX[index].bus_higher, a_inX[index].bus_lower,
           b_inX[index].bus_higher, b_inX[index].bus_lower);
    index ++;
  }

  printf("\nnow scanning y-coord ...\n");
  index=0;
  while((fscanf(inp2, "%x %x %x %x\n",
                &(a_inY[index]).bus_higher,
&(a_inY[index]).bus_lower,
                &(b_inY[index]).bus_higher,
&(b_inY[index]).bus_lower))==4){

    printf("...slice %d: 1st pt -> %x%x; 2nd point -> %x%x\n",
           index,
           a_inY[index].bus_higher, a_inY[index].bus_lower,
           b_inY[index].bus_higher, b_inY[index].bus_lower);
    index ++;
  }

  printf("\nnow scanning z-coord ...\n");
  index=0;
  while((fscanf(inp3, "%x %x %x %x\n",
                &(a_inZ[index]).bus_higher,
&(a_inZ[index]).bus_lower,
                &(b_inZ[index]).bus_higher,
&(b_inZ[index]).bus_lower))==4){

    printf("...slice %d: 1st pt -> %x%x; 2nd point -> %x%x\n",
           index,
           a_inZ[index].bus_higher, a_inZ[index].bus_lower,
           b_inZ[index].bus_higher, b_inZ[index].bus_lower);
    index ++;
  }

  printf("\nnow scanning curvedef1 ...\n");
  index=0;
  while((fscanf(inp4, "%x %x\n",
                &(curve_def1[index]).bus_higher,
&(curve_def1[index]).bus_lower))==2){

    printf("%x%x\n",
             curve_def1[index].bus_higher,
curve_def1[index].bus_lower);

    index ++;
  }
  printf("\nnow scanning curvedef2 ...\n");
  index=0;
  while((fscanf(inp5, "%x %x\n",
                &(curve_def2[index]).bus_higher,
&(curve_def2[index]).bus_lower))==2){

    printf("%x%x\n",
             curve_def2[index].bus_higher,
curve_def2[index].bus_lower);
    index ++;
  }
  printf("SCANNED...\n");
fclose(inp1);
fclose(inp2);
fclose(inp3);
fclose(inp4);
fclose(inp5);
fclose(outp1);

}

/*****************************************************************/
/*****************************************************************/
```

```
void scan_multiplier(int multiplier[SLICE])
{
  int index;
  FILE *inp;
  index = SLICE-1;
  if((inp = fopen("mult_int", "r")) == NULL)
    {
      printf("Open Failed\n");
      /*end(1);*/
    }

  printf("\nnow scanning curve point multiplier ...\n");
  while((((fscanf(inp, "%x\n", &multiplier[index]))==1) && (index > -1))){
    printf("...slice %d: hex rep. -> %x\n", index, multiplier[index]);
    index --;
  }

}

/*****************************************************************/
/*****************************************************************/
void print_vector(cmp_bus vector[SLICE])
{
  int index;
  for(index=SLICE-1; index>-1; index--){
    printf("%x%x",
           vector[index].bus_higher, vector[index].bus_lower);
  }
}

/*****************************************************************/
/*****************************************************************/
void curve_mult(cmp_bus a_inX[SLICE],
cmp_bus a_inY[SLICE], cmp_bus a_inZ[SLICE],
              cmp_bus curve_def1[SLICE], cmp_bus curve_def2[SLICE],
              cmp_bus res_outX[SLICE], cmp_bus res_outY[SLICE],
cmp_bus res_outZ[SLICE], int multiplier[SLICE])
{
  cmp_bus tmpX[SLICE];
  cmp_bus tmpY[SLICE];
  cmp_bus tmpZ[SLICE];
  int i,j;
  int outer_bound, inner_bound;
  int mask = 0x01;
  int first = 1;
  int double_count = 0;
  int add_count = 0;

  copy_vector(a_inX, tmpX);
  copy_vector(a_inY, tmpY);
  copy_vector(a_inZ, tmpZ);

  /*finds the first bit in the 168 bit integer*/
  scan_array(multiplier, &outer_bound, &inner_bound);

  /*start from the next bit after the one found in previous function*/
  inner_bound--;

  /*adjust mask to that bit position*/
  mask = mask<<(inner_bound-1);

  /*start S-A-M */
  for(i=outer_bound; i>-1; i--){

    if(first == 0){
      inner_bound = SLICE_WIDTH;
      mask = 0x80;
    } else {
      first = 0;
    }

    for(j=inner_bound; j>0; j--){
      printf("******************\n");
      printf("******************\n");
      printf("Results for bit %d:\n", (i*(8))+j);
      printf("i =  %d:\n", i);
      printf("INPUT:\n");
      print_vector(a_inX);
      printf("\n");
      print_vector(a_inY);
      printf("\n");
      print_vector(a_inZ);
      printf("\n");
      pt_double(a_inX, a_inY, a_inZ,
                curve_def1,
                res_outX, res_outY, res_outZ);

      copy_vector(res_outX, a_inX);
      copy_vector(res_outY, a_inY);
      copy_vector(res_outZ, a_inZ);
      printf("...DOUBLE...\n");
      double_count++;
      print_vector(res_outX);
      printf("\n");
      print_vector(res_outY);
      printf("\n");
      print_vector(res_outZ);
      printf("\n");
```

```
      if((mask & multiplier[i]) != 0){
        printf("INPUT:\n");
        print_vector(a_inX);
        printf("\n");
        print_vector(a_inY);
        printf("\n");
        print_vector(a_inZ);
        printf("\n");
        print_vector(tmpX);
        printf("\n");
        print_vector(tmpY);
        printf("\n");
        pt_add(a_inX, a_inY, a_inZ,
               tmpX, tmpY, tmpZ,
               curve_def2,
               res_outX, res_outY, res_outZ);
        copy_vector(res_outX, a_inX);
        copy_vector(res_outY, a_inY);
        copy_vector(res_outZ, a_inZ);
        printf("...ADD...\n");
        print_vector(res_outX);
        printf("\n");
        print_vector(res_outY);
        printf("\n");
        print_vector(res_outZ);
        printf("\n");
        add_count++;
      }
      mask=mask>>1;
    }
  }
  copy_vector(a_inX, res_outX);
  copy_vector(a_inY, res_outY);
  copy_vector(a_inZ, res_outZ);

  printf("Double operations performed = %d\n", double_count);
  printf("Add operations performed = %d\n", add_count);
}

/*******************************************************************/
/*******************************************************************/
void scan_array(int multiplier[SLICE],
int *outer_index,
int *inner_index)
{
  int i = SLICE;
  int j;
  int mask = 0x80;
  int empty = 1;

  while((i>-1) && (empty==1)){
    j = SLICE_WIDTH;
    i--;
    if((multiplier[i] & mask) == 0){
      empty = 1;
    } else {
      empty = 0;
    }
    while((empty==1) && (j>0)){
      j--;
    }
    mask = mask>>1;
  }
  *outer_index = i;
  *inner_index = j;
}
```

Appendix C

Synosys Script

```
/* Sample Script for Synopsys to Xilinx Using */
/* FPGA Compiler targeting a XC4000EX device */
/* Set the name of the design"s top-level */
        TOP = system
        F1 = packages
        F2 = dpmem
F3 = reg
F4 = reg_le

F5 = add2
F6 = add3
F7 = bit_add
        F8 = mult2k4
        F9 = madd4
        F10 = const

F11 = mult
F12 = DOUBLE_fsm
F13 = ADD_fsm
        F14 = IO_fsm
F15 = reg_bank
F16 = SWITCH1
F17 = SWITCH2
F18 = SWITCH3
F19 = SWITCH4
F20 = SWITCH5
F21 = add_array_lower
F22 = add_bank
F23 = HYBMULT

        designer = "Martin Rosner"
        company  = "WPI Crypto Group"
        part     = "4062XLPG475-1"

/* Analyze and Elaborate the design file.       */
        analyze -format vhdl "vhdl/" + F1 + ".vhd"

        read -format edif "WORK/" + F2 + ".edn"
        read -format edif "WORK/" + F3 + ".edn"
        read -format edif "WORK/" + F4 + ".edn"

        analyze -format vhdl "vhdl/" + F5 + ".vhd"
        analyze -format vhdl "vhdl/" + F6 + ".vhd"
        analyze -format vhdl "vhdl/" + F7 + ".vhd"
        analyze -format vhdl "vhdl/" + F8 + ".vhd"
        analyze -format vhdl "vhdl/" + F9 + ".vhd"
        analyze -format vhdl "vhdl/" + F10 + ".vhd"
        analyze -format vhdl "vhdl/" + F11 + ".vhd"
        analyze -format vhdl "vhdl/" + F12 + ".vhd"
        analyze -format vhdl "vhdl/" + F13 + ".vhd"
        analyze -format vhdl "vhdl/" + F14 + ".vhd"
        analyze -format vhdl "vhdl/" + F15 + ".vhd"
        analyze -format vhdl "vhdl/" + F16 + ".vhd"
        analyze -format vhdl "vhdl/" + F17 + ".vhd"
        analyze -format vhdl "vhdl/" + F18 + ".vhd"
        analyze -format vhdl "vhdl/" + F19 + ".vhd"
        analyze -format vhdl "vhdl/" + F20 + ".vhd"
        analyze -format vhdl "vhdl/" + F21 + ".vhd"
        analyze -format vhdl "vhdl/" + F22 + ".vhd"
        analyze -format vhdl "vhdl/" + F23 + ".vhd"
        analyze -format vhdl "vhdl/" + TOP + ".vhd"

        elaborate TOP


/* Set the current design to the reg_bank level.        */
        current_design F15
/* Don't touch the logiblox*/
        set_dont_touch (bank_i_*)

/* Set the current design to the HYBMULT level.        */
        current_design F23
/* Don't touch the logiblox*/
        set_dont_touch (BANK_REG_LE_*)
        set_dont_touch (BANK_REG_*)

/* Set the current design to the top level.        */
        current_design TOP
remove_constraint -all
/* Uniquify the design and reset the schematic */
```

```
        uniquify
        create_schematic -size infinite -gen_database

/* include timming and area contraints */
create_clock clk -period 100
set_input_delay 0 -clock clk { all_inputs()}
set_output_delay 0 -clock clk { all_outputs()}
set_operating_conditions WCCOM

/* Indicate which ports are pads.    */
        set_port_is_pad "*"

set_pad_type -no_clock all_inputs()
set_pad_type -clock clk
        set_pad_type -slewrate LOW all_outputs()
        insert_pads

/* link */
link

/* Synthesize the design.*/
/*        compile -map_effort high -ungroup_all*/
    compile -boundary_optimization -map_effort high

/* Write the design report files.                 */
        sh rm -f "reports/" + TOP + ".old"
        sh cat "reports/" + TOP + ".fpga" "reports/" + TOP + ".timing"\
 "reports/" + TOP + ".cnst" > "reports/" + TOP + ".old"
        report_fpga > "reports/" + TOP + ".fpga"
        report_timing > "reports/" + TOP + ".timing"
        report_constraint -verbose > "reports/" + TOP + ".cnst"

/* Write out an intermediate DB file to save state */
        write -format db -hierarchy -output "db/" + TOP + "_compiled.db"

/* Replace CLBs and IOBs primitives (XC4000E/EX/XL only)    */
        replace_fpga

/* Set the part type for the output netlist.      */
        set_attribute TOP "part" -type string part


/* Write out the intermediate DB file to save state*/
        write -format db -hierarchy -output "db/" + TOP + ".db"

/* Write out the timing constraints               */
        ungroup -all -flatten
        write_script > "dc/" + TOP + ".dc"
```

```
/* Save design in XNF format as <design>.sxnf      */
        write -format xnf -hierarchy -output "sxnf/" + TOP + ".sxnf"

/* XILINX primitive to convert Synopsys design constraints to Xilinx format*/
sh dc2ncf "dc/" + TOP + ".dc"
```

Appendix D

Simulation Results

D.1 $GF((2^4)^9)$ Multilication Simulation

The next figure shows the short simulation of one Galois field multiplication. two inputs to this hybrid multiplier are `a_op_S1` and `b_op_S1`. The output is `mult_S2`. `double_fsm_ld_gfm` is used to load enable the internal registers on the hybrid multiplier.

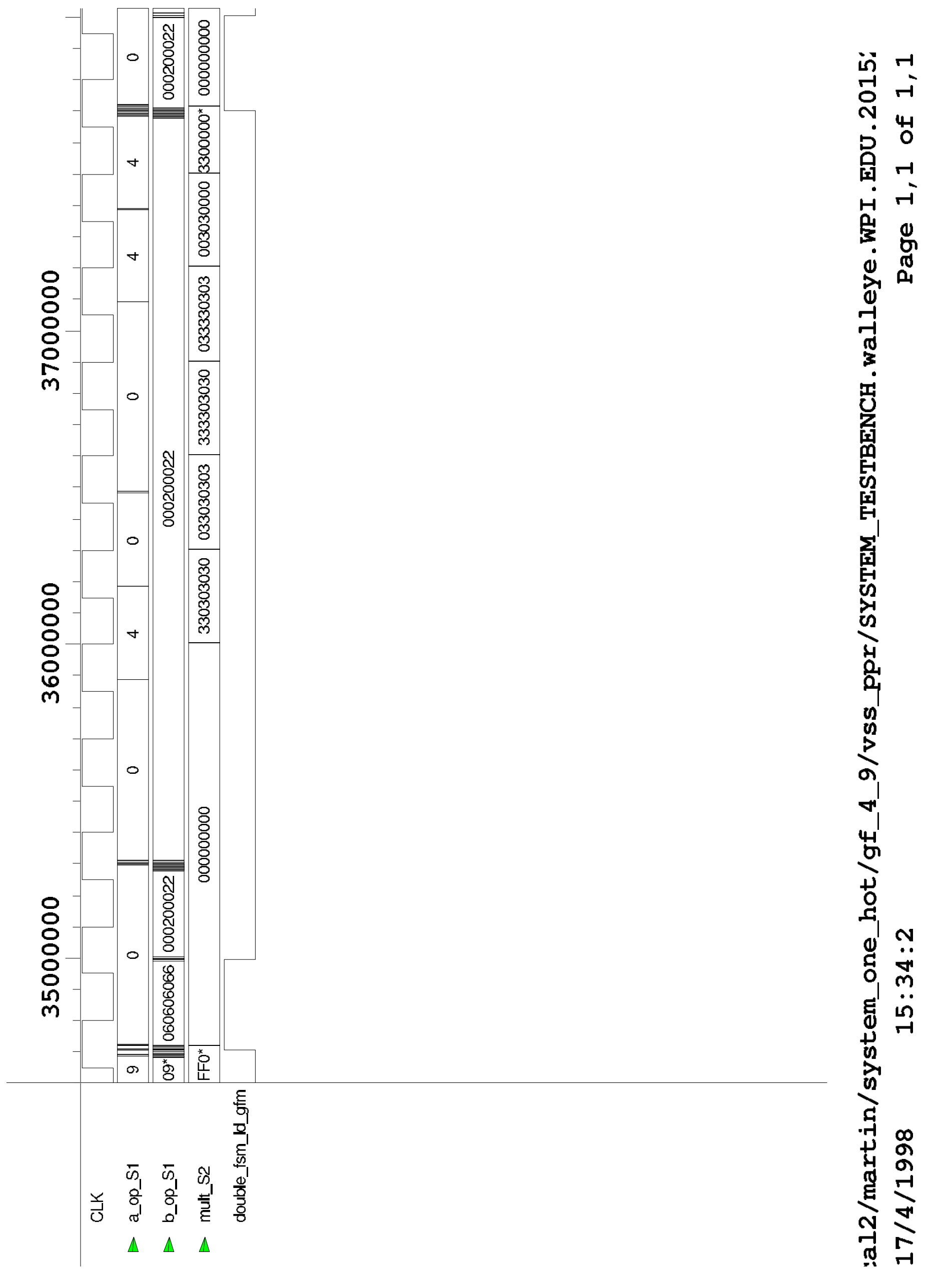

Figure D.1: $GF((2^4)^9)$ multiplication

D.2 $GF((2^4)^9)$ Double Sequance

The following sequence of six figures shows the post place-and-route simulation results for the double group operation. The signals shown in the simulation sequence are as follows:

`clk` ⇒ clock signal.

`a_op_S1` ⇒ switch 1 output1.

`b_op_S1` ⇒ switch 1 output2.

`mult_S2` ⇒ GF multiplier result.

`double_fsm_lg_gfm` ⇒ GF multiplier load control signal from double state machine.

`double_fsm_mux1_a()` ⇒ switch 1 control signal from double state machine.

`double_fsm_mux2_add` ⇒ switch 2 control signal from double state machine.

`double_fsm_wrt_en` ⇒ write enable control signal from double state machine.

`dpo_S1` ⇒ second RAM output port.

`ram_read_addr()` ⇒ memory read address.

`ram_write_addr()` ⇒ memory write/read address.

`ram_wrt_en` ⇒ memory write enable.

`s2_DI` ⇒ memory input data port.

`done_double_io` ⇒ asserted when group double is done.

`IO_FSM_COMP_MACHIN...` ⇒ current state of I/O state machine.

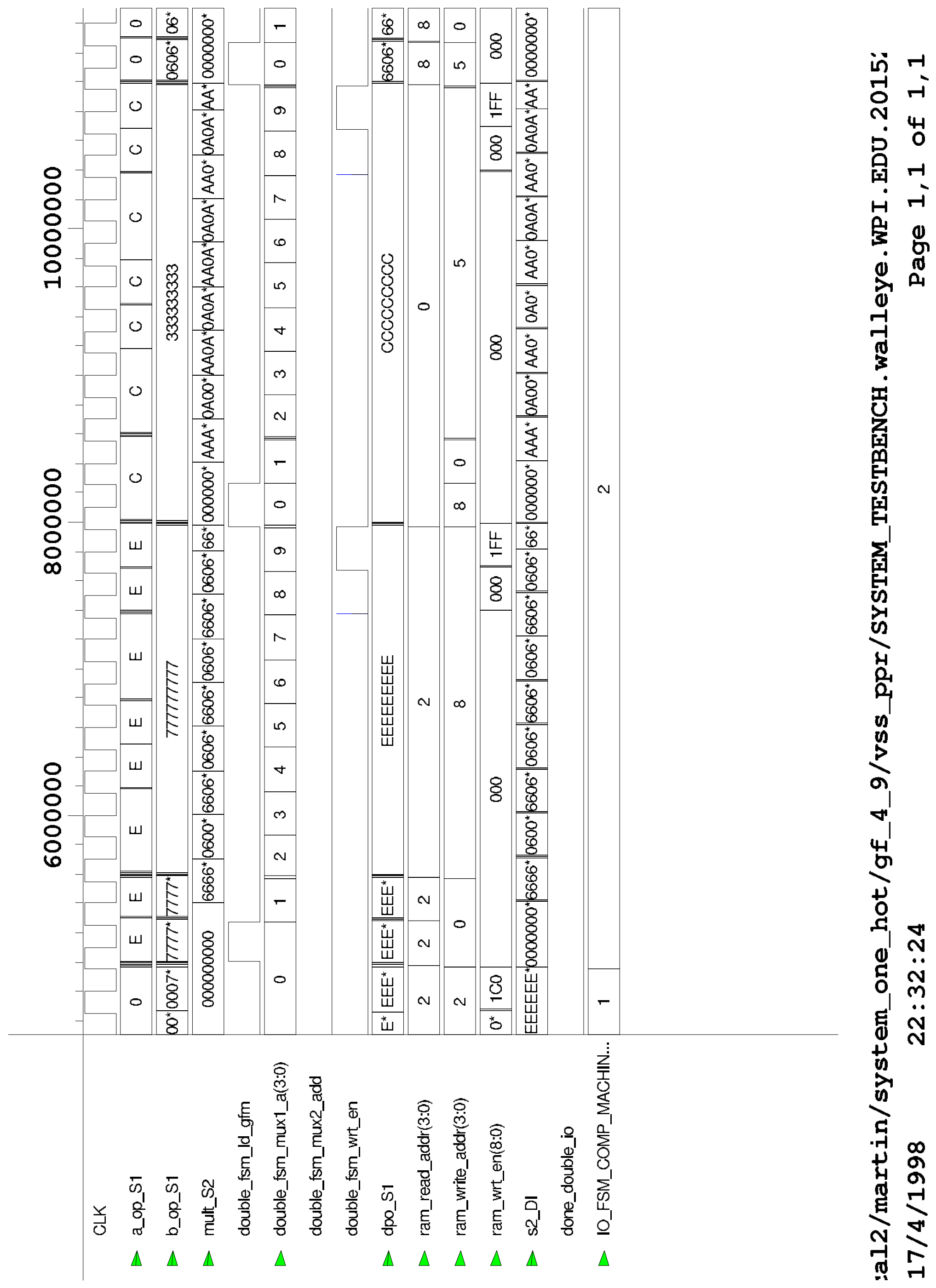

Figure D.2: $GF((2^4)^9)$ double sequance

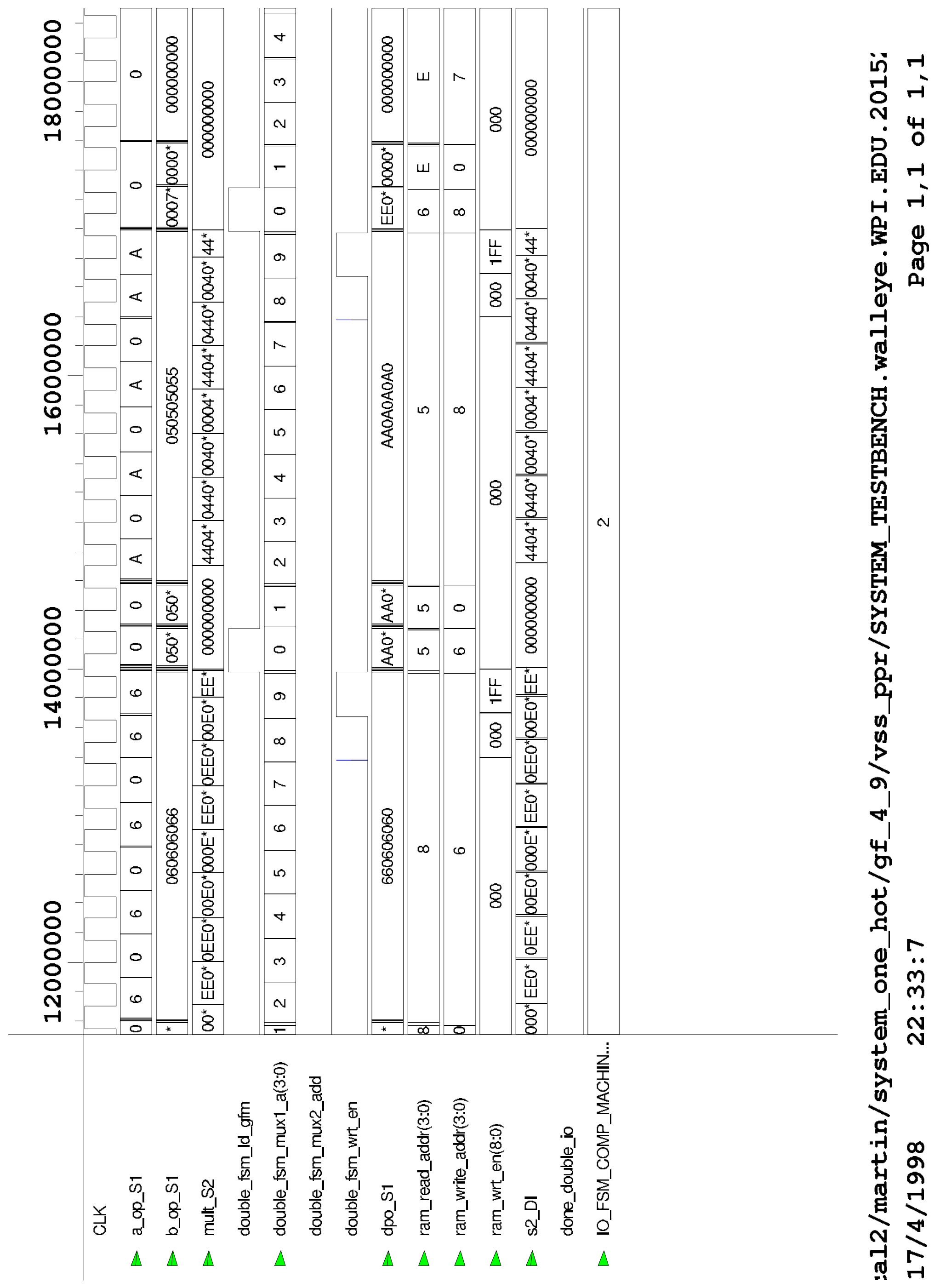

Figure D.3: $GF((2^4)^9)$ double sequance (cont.)

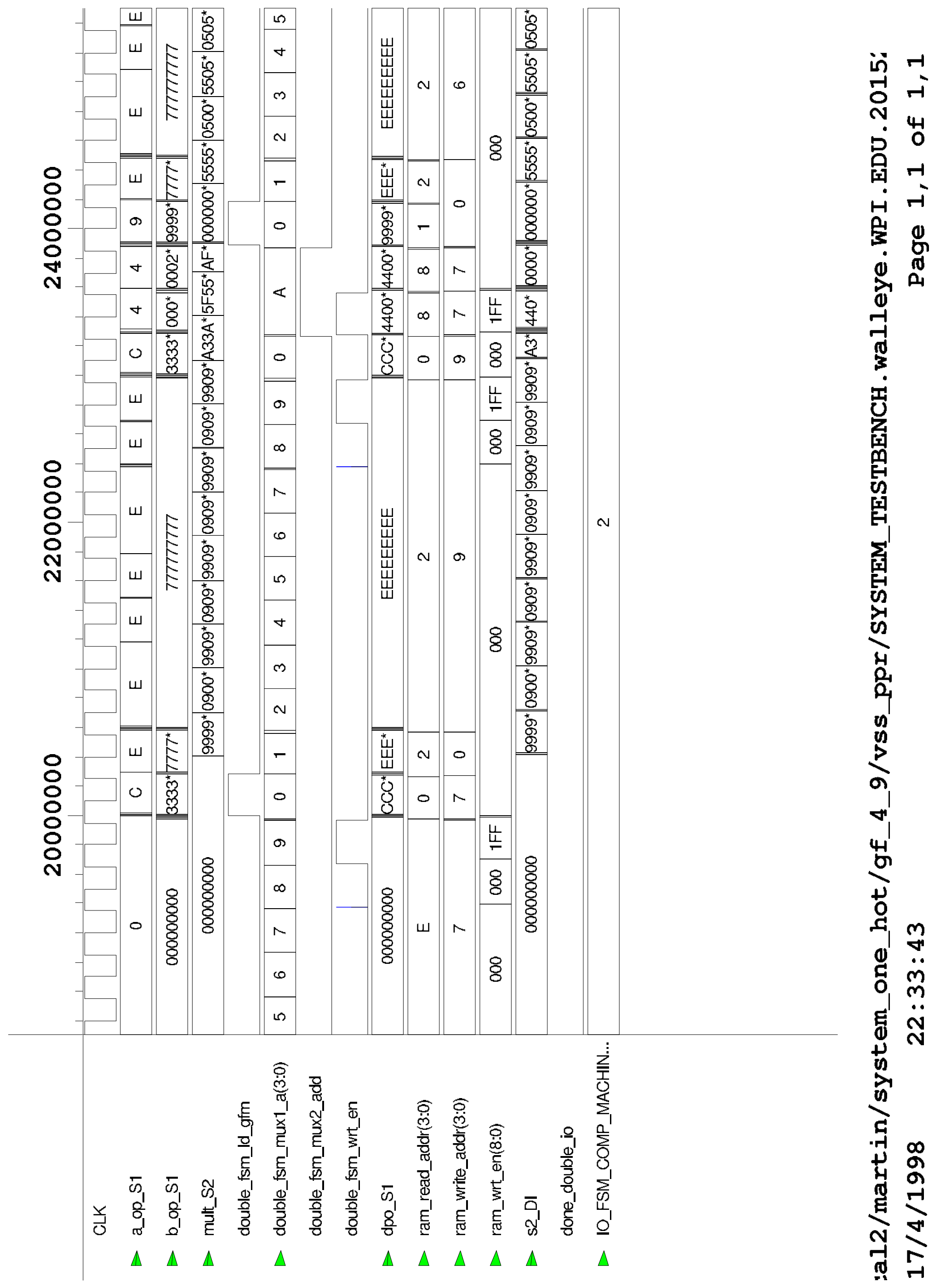

Figure D.4: $GF((2^4)^9)$ double sequance (cont.)

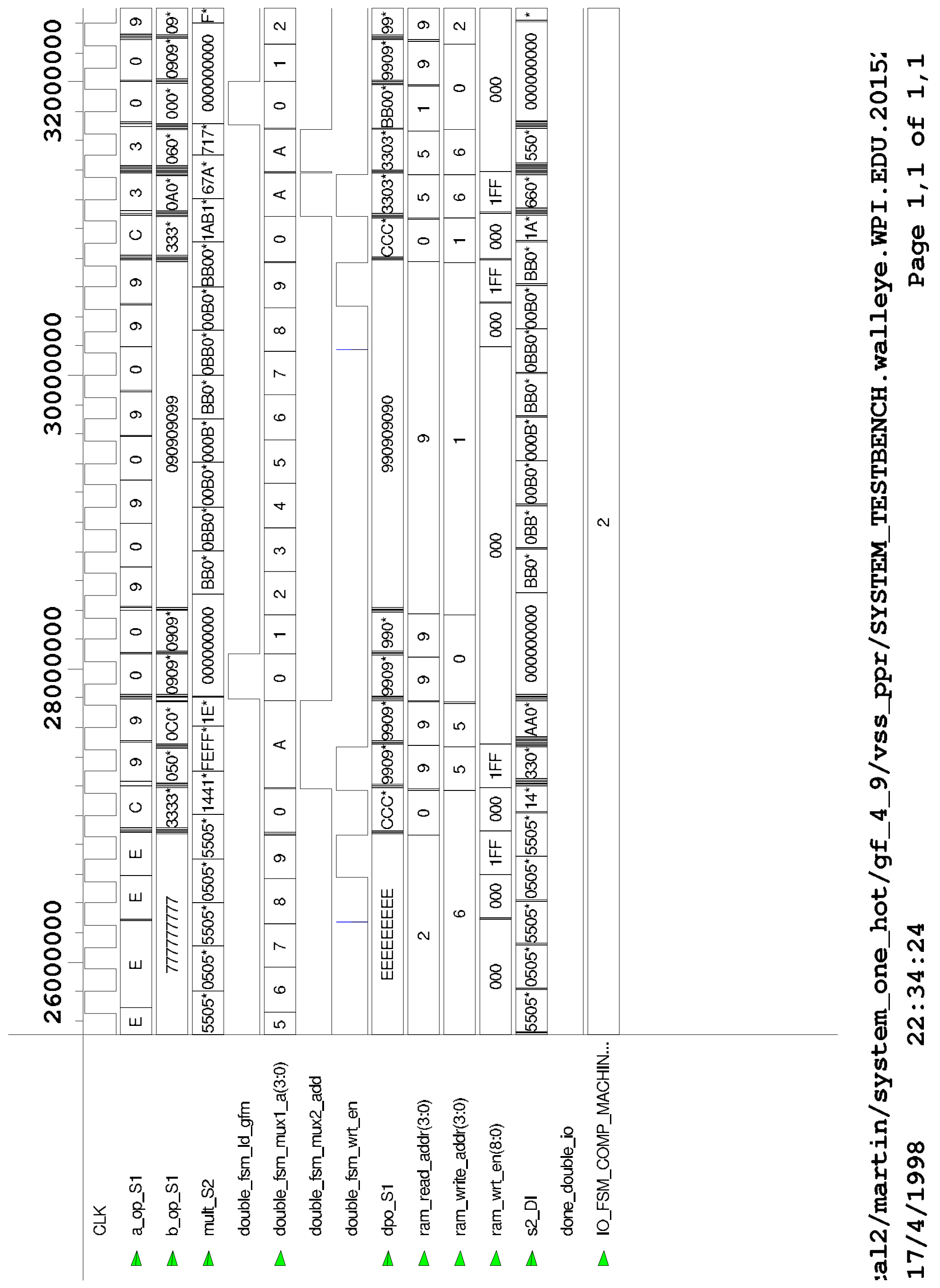

Figure D.5: $GF((2^4)^9)$ double sequance (cont.)

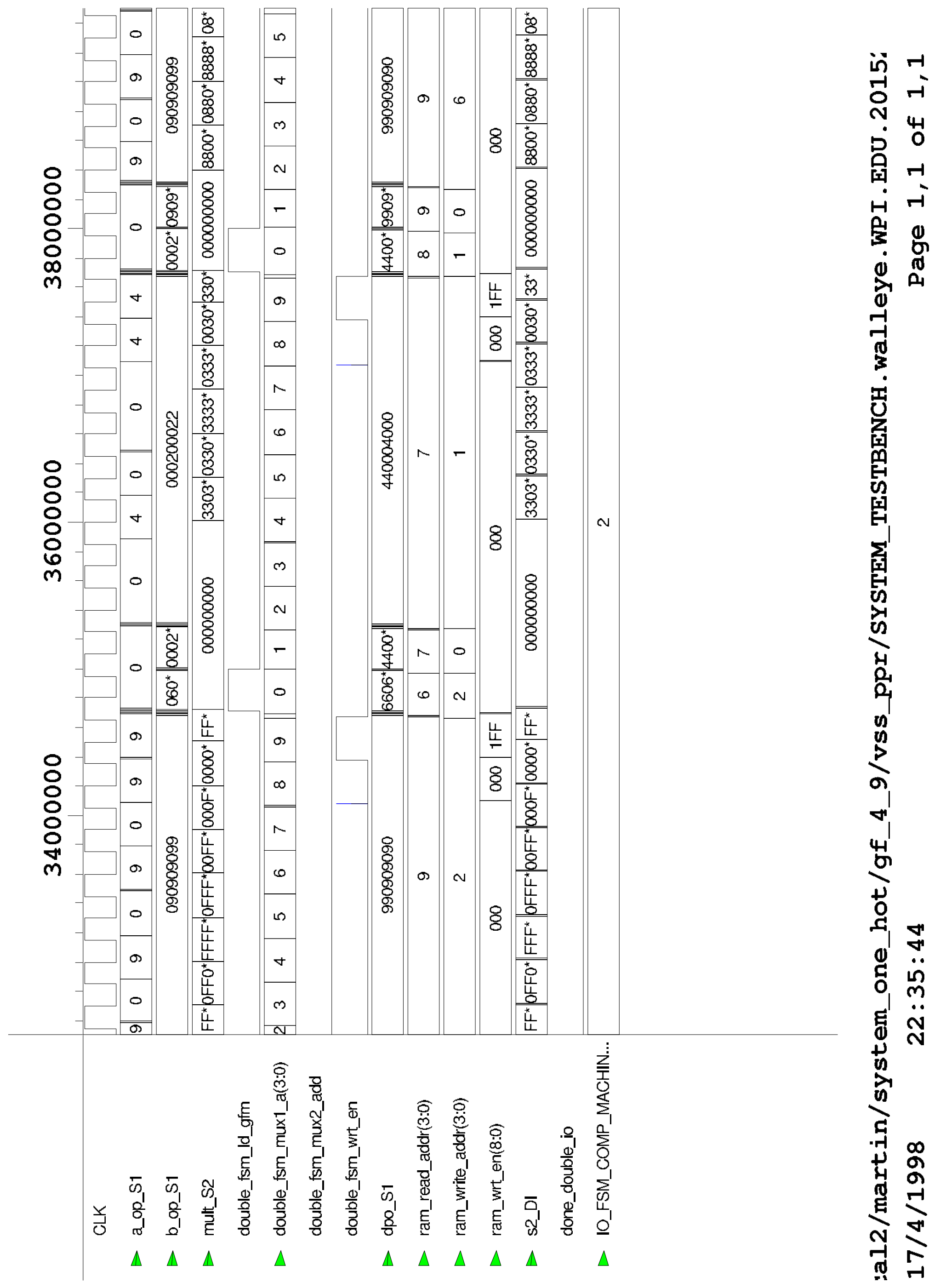

Figure D.6: $GF((2^4)^9)$ double sequance(cont.)

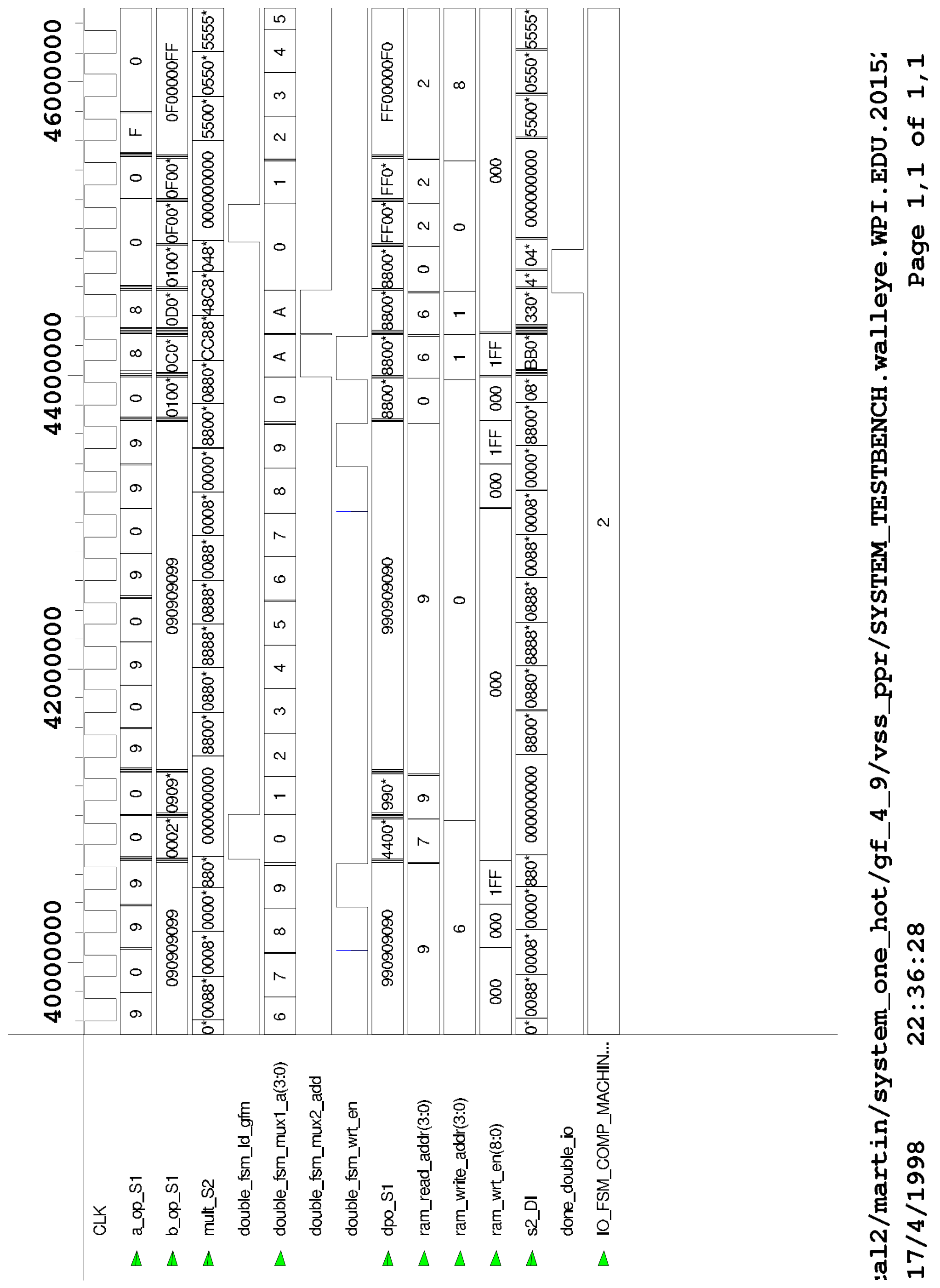

Figure D.7: $GF((2^4)^9)$ double sequance (cont.)

D.3 $GF((2^4)^9)$ Add Sequance

The following sequence of eight figures shows the post place-and-route simulation results for the add group operation. The signals shown in the simulation sequence are as follows:

`clk` ⇒ clock signal.

`a_op_S1` ⇒ switch 1 output1.

`b_op_S1` ⇒ switch 1 output2.

`mult_S2` ⇒ GF multiplier result.

`double_fsm_lg_gfm` ⇒ GF multiplier load control signal from double state machine.

`double_fsm_mux1_a()` ⇒ switch 1 control signal from double state machine.

`double_fsm_mux2_add` ⇒ switch 2 control signal from double state machine.

`double_fsm_wrt_en` ⇒ write enable control signal from double state machine.

`dpo_S1` ⇒ second RAM output port.

`ram_read_addr()` ⇒ memory read address.

`ram_write_addr()` ⇒ memory write/read address.

`ram_wrt_en` ⇒ memory write enable.

`s2_DI` ⇒ memory input data port.

`done_double_io` ⇒ asserted when group double is done.

`IO_FSM_COMP_MACHIN...` ⇒ current state of I/O state machine.

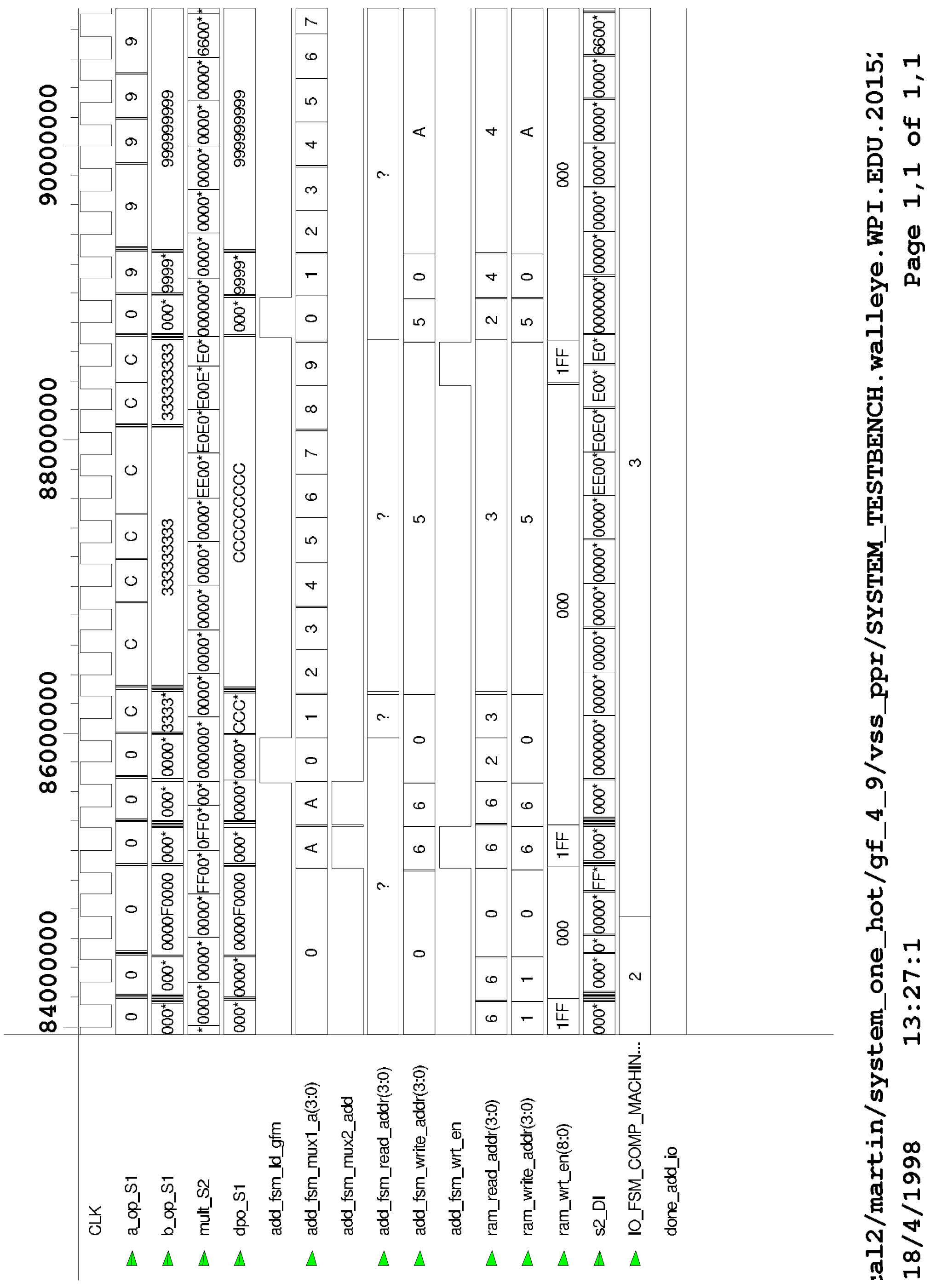

Figure D.8: $GF((2^4)^9)$ add sequance

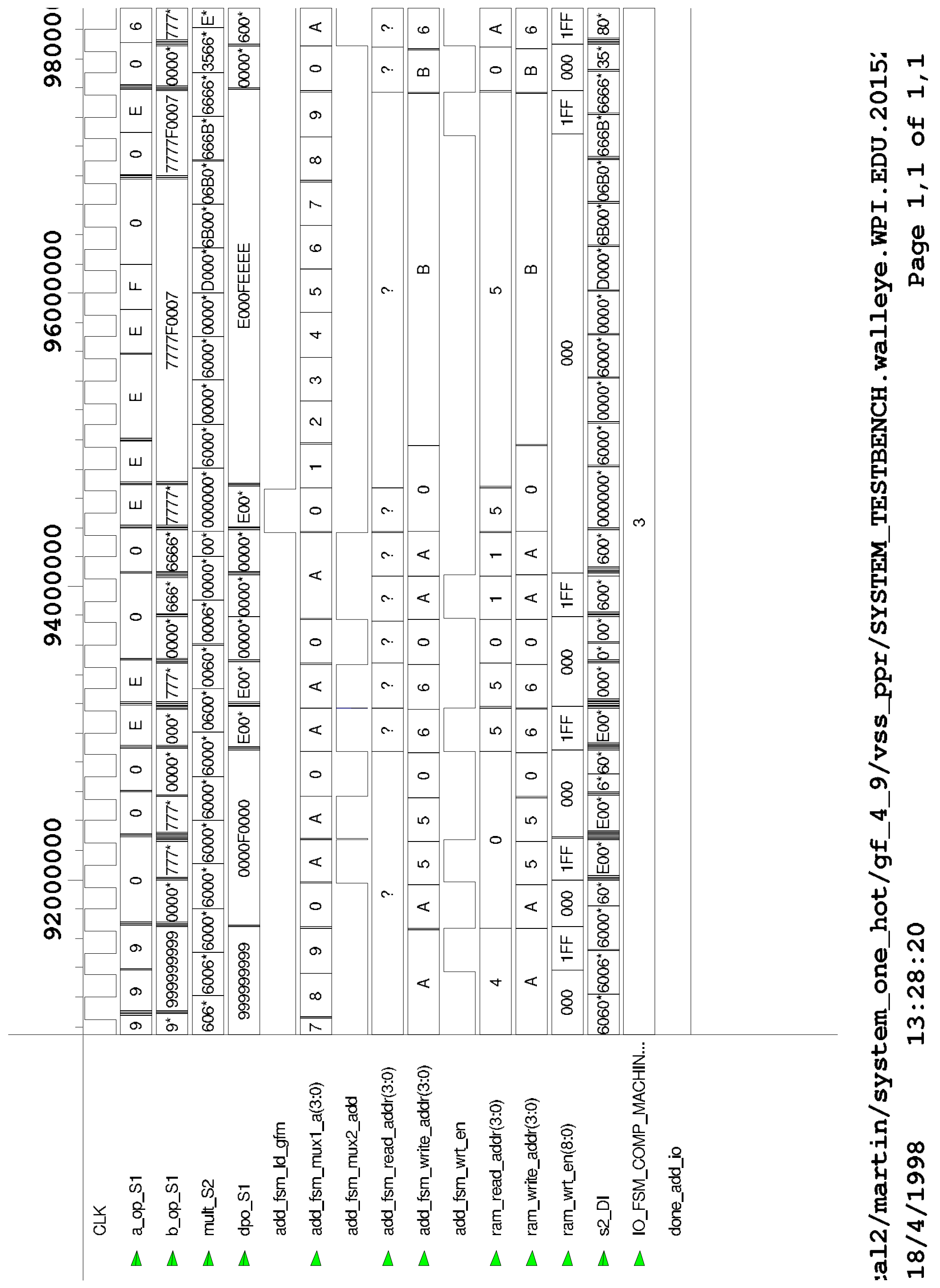

Figure D.9: $GF((2^4)^9)$ add sequance (cont.)

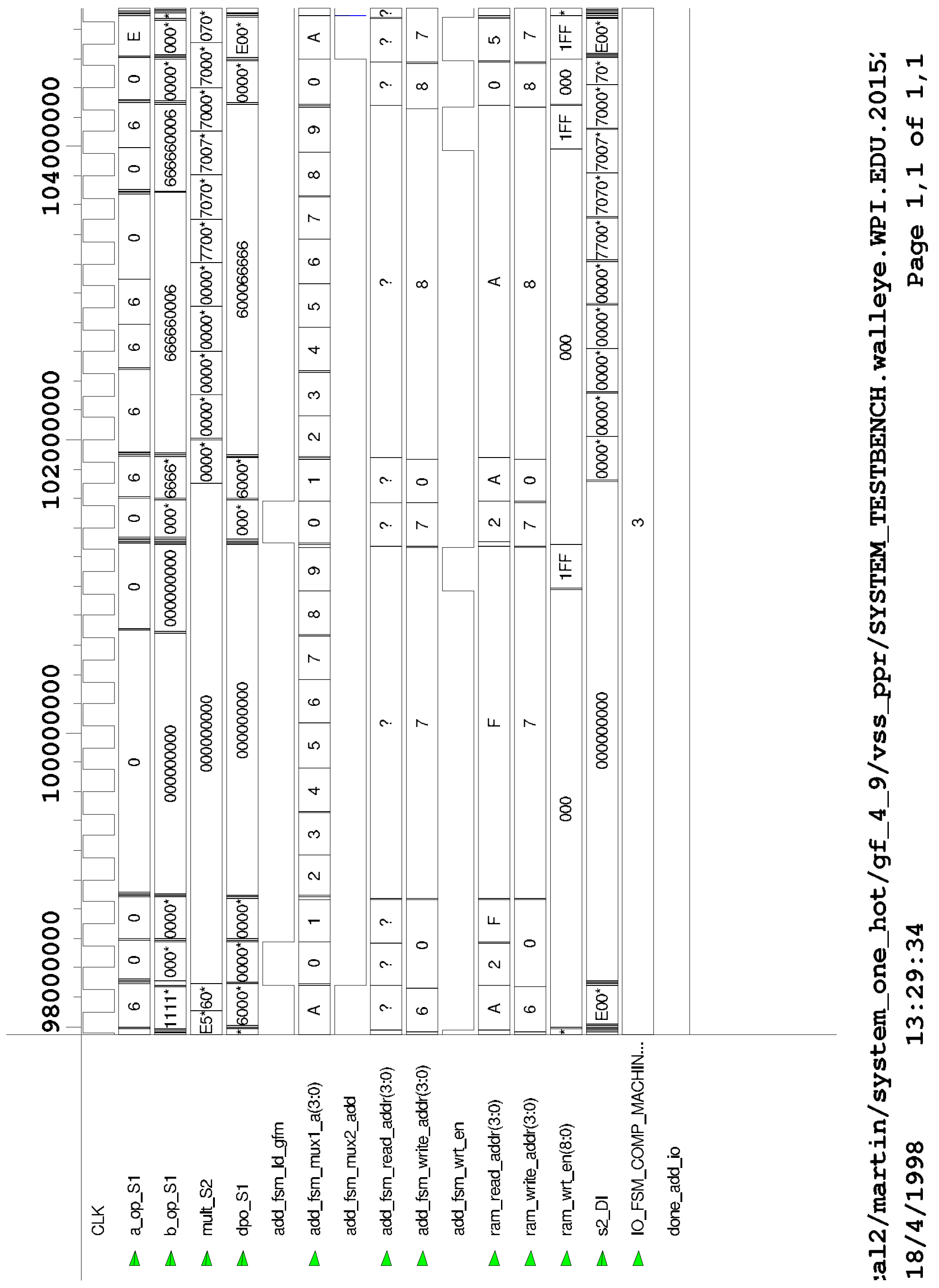

Figure D.10: $GF((2^4)^9)$ add sequance (cont.)

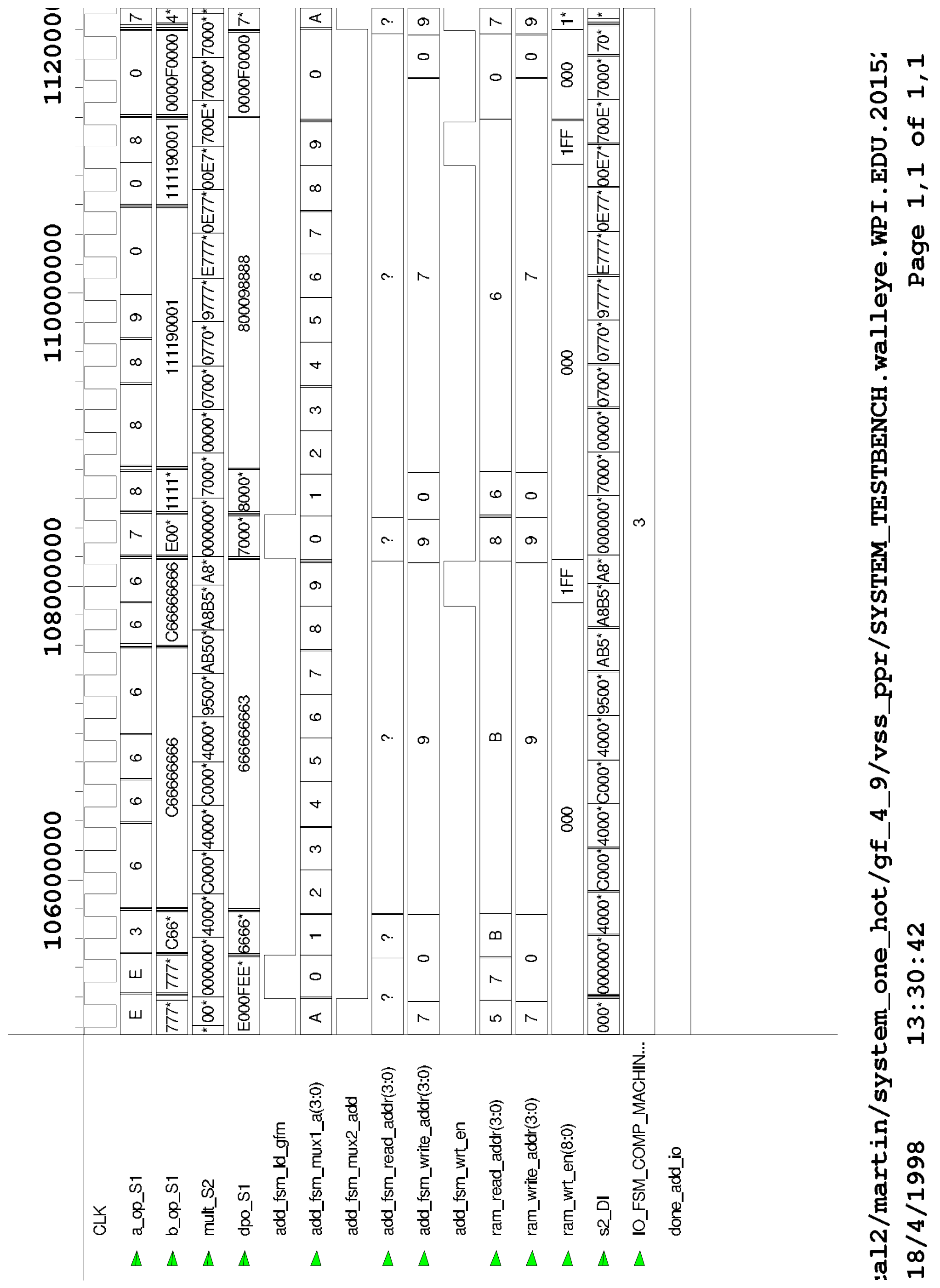

Figure D.11: $GF((2^4)^9)$ add sequance (cont.)

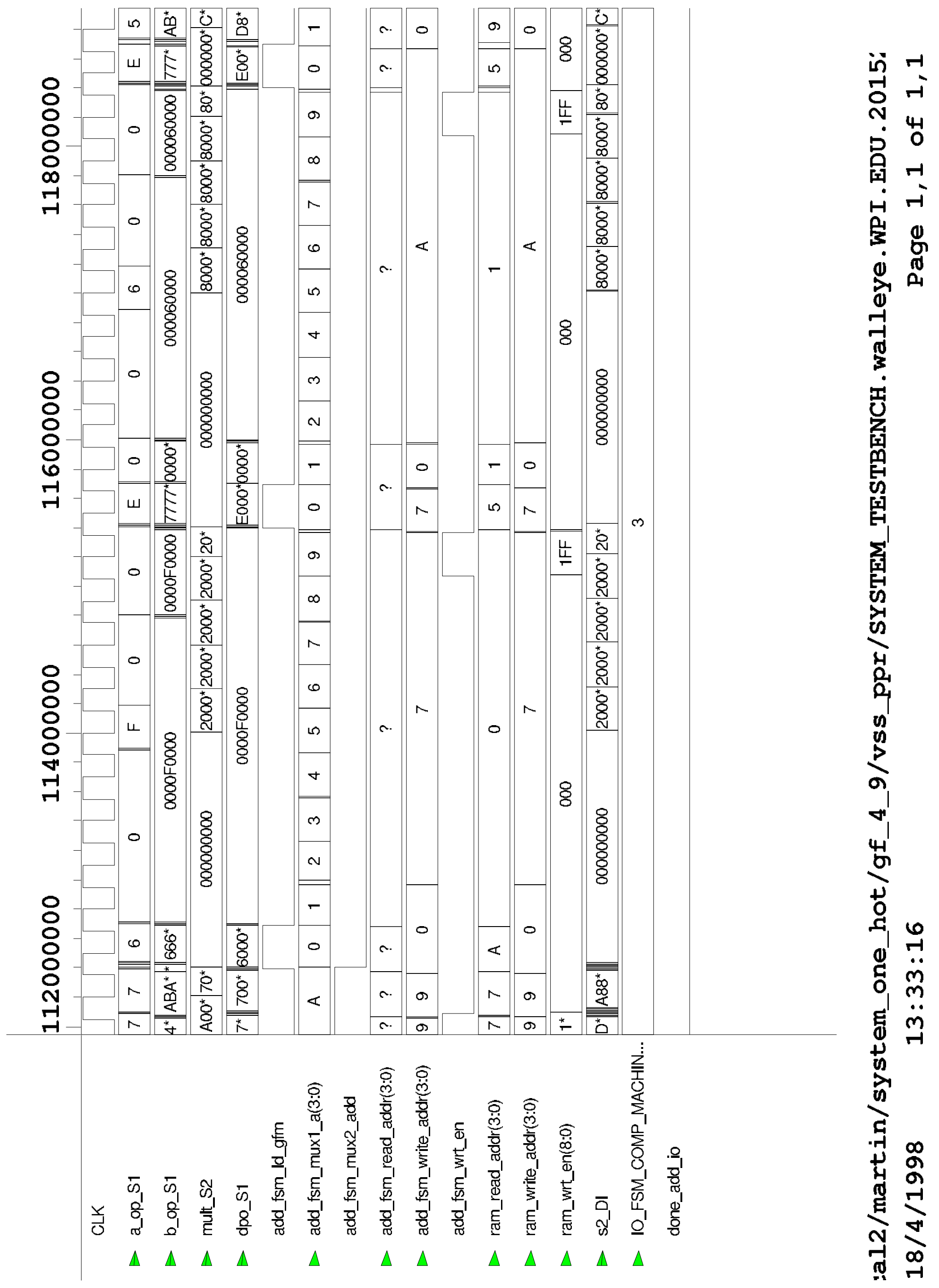

Figure D.12: $GF((2^4)^9)$ add sequance (cont.)

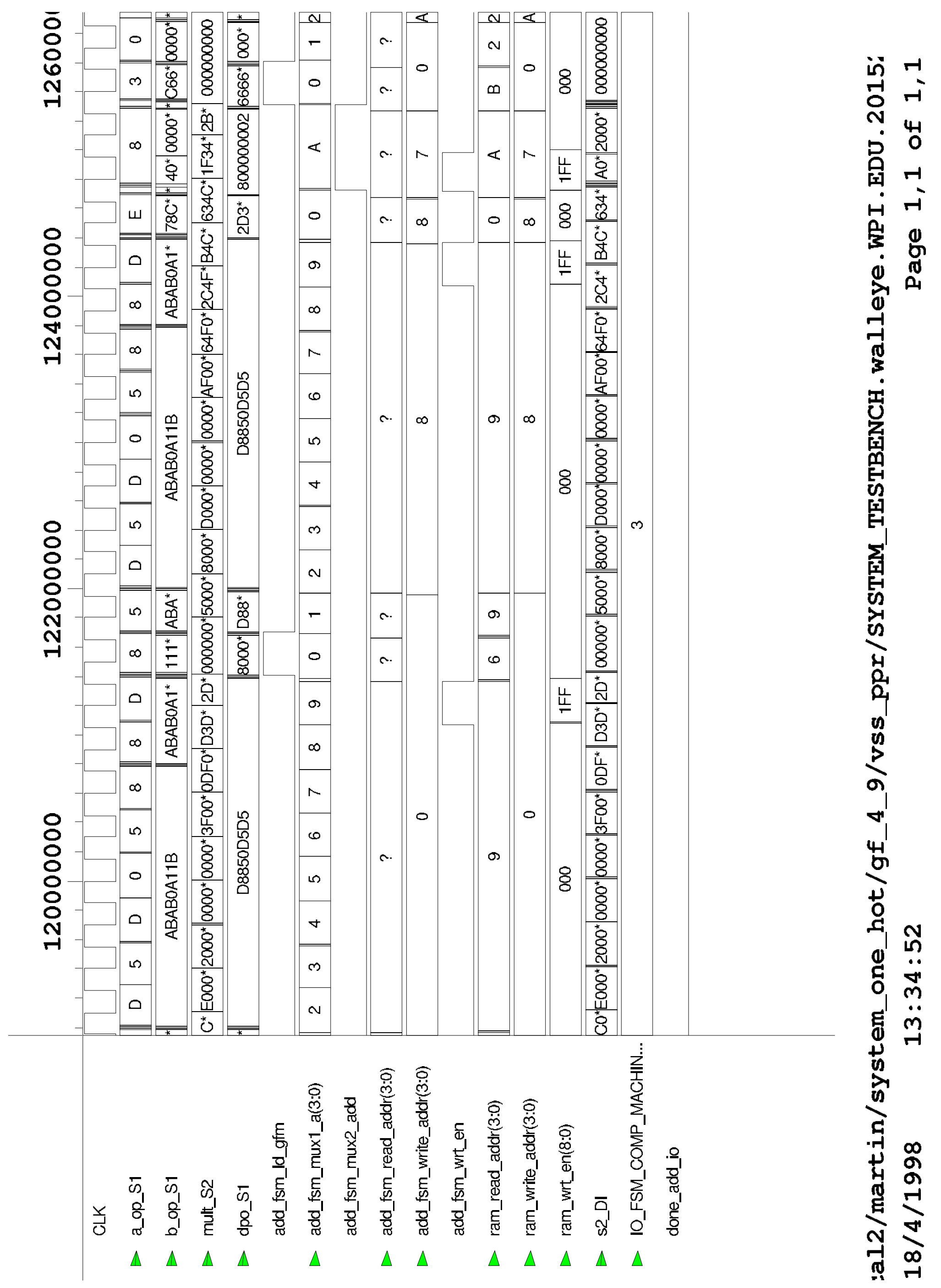

Figure D.13: $GF((2^4)^9)$ add sequance (cont.)

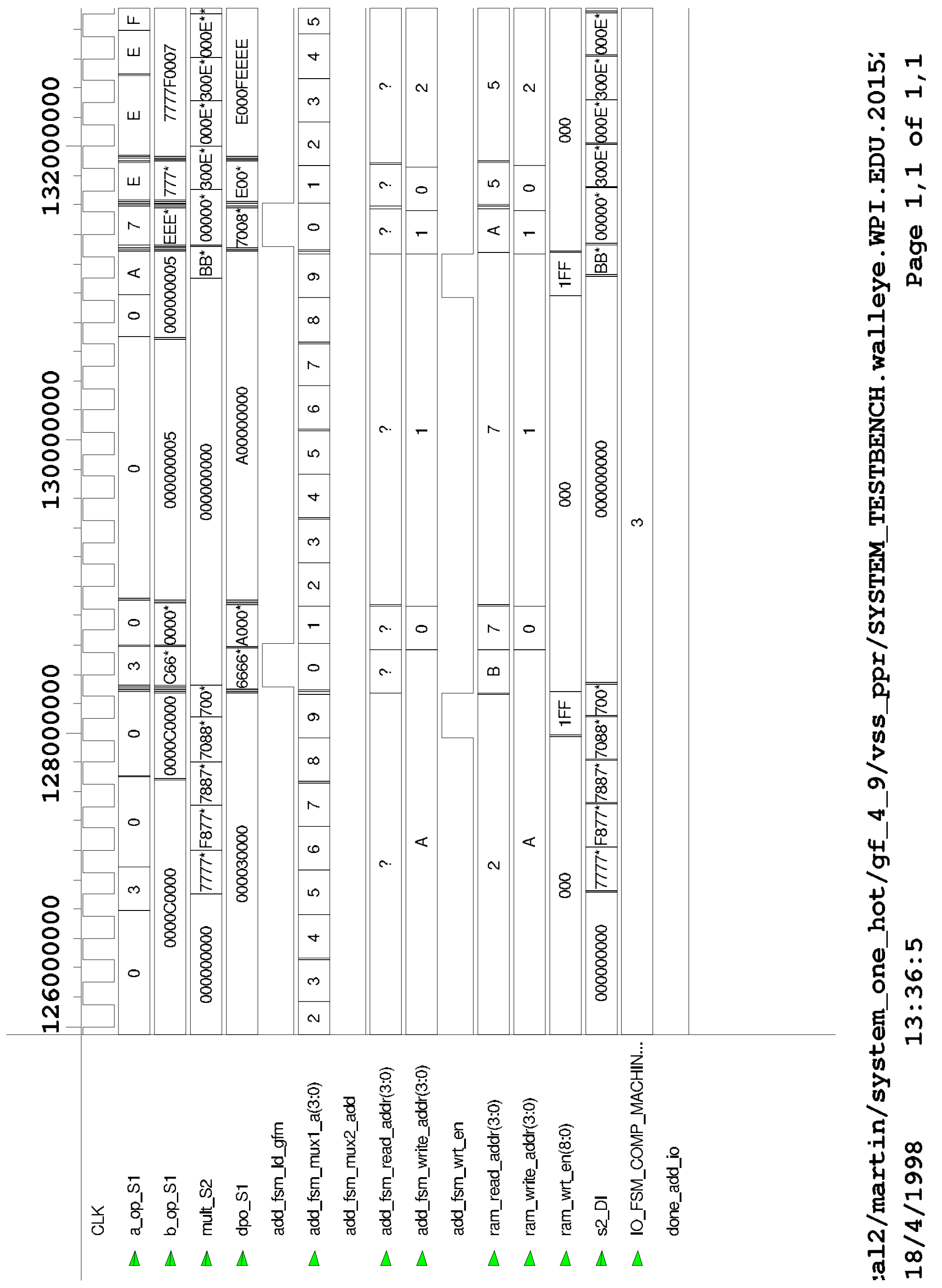

Figure D.14: $GF((2^4)^9)$ add sequance (cont.)

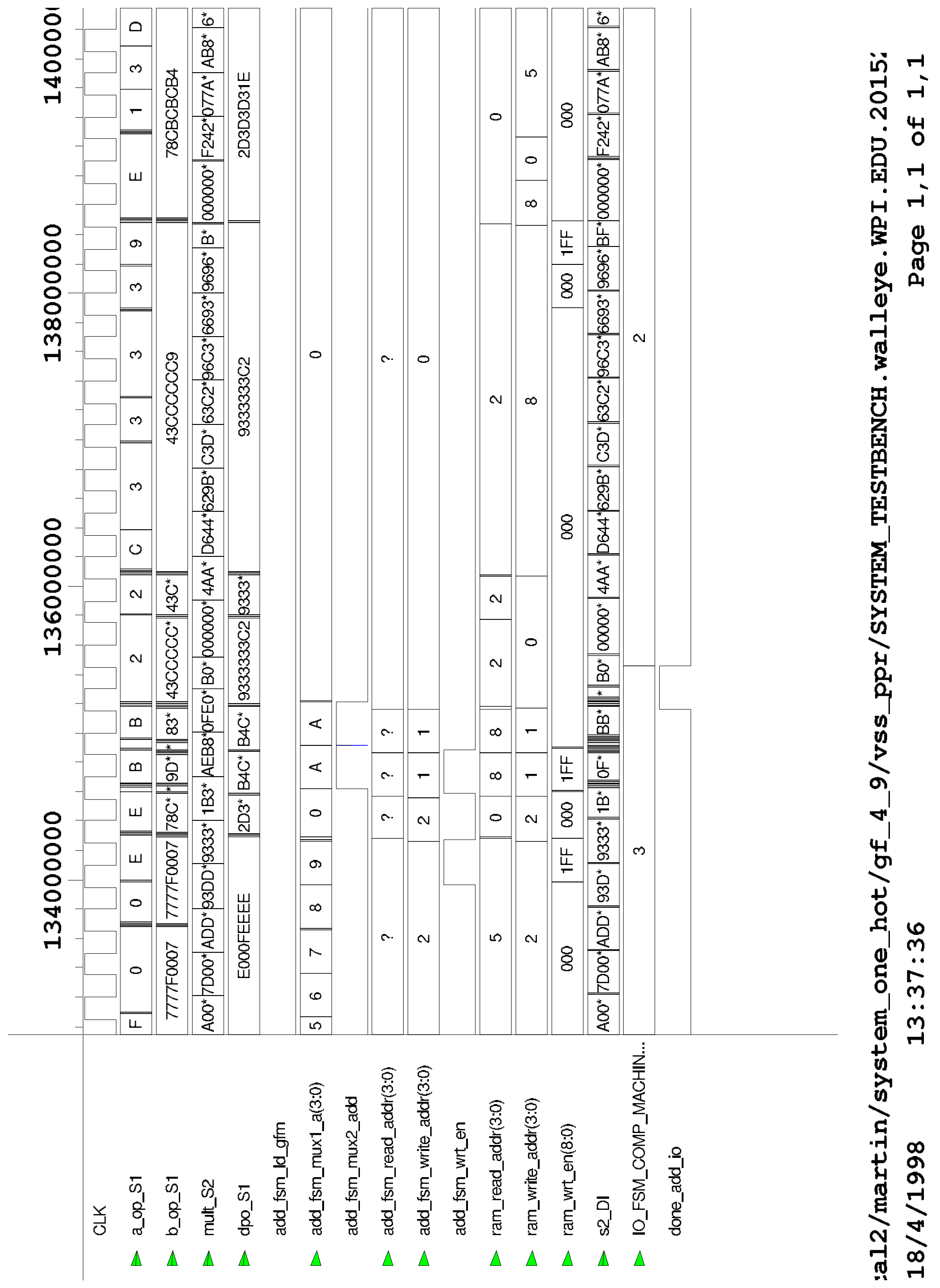

Figure D.15: $GF((2^4)^9)$ add sequance (cont.)

Bibliography

[1] IEEE P1363 Standard Specifications For Public-Key Cryptography. Draft, release expected in 1998.

[2] V.B. Afanasyev. Complexity of VLSI implementation of finite field arithmetic. In *II. Intern. Workshop on Algebraic and Combinatorial Coding Theory*, pages 6–7, Leningrad, USSR, September 1990.

[3] V.B. Afanasyev. On the complexity of finite field arithmetic. In *5th Joint Soviet-Swedish Intern. Workshop on Information Theory*, pages 9–12, Moscow, USSR, January 1991.

[4] G.B. Agnew, R.C. Mullin, I.M. Onyschuk, and S.A. Vanstone. An implemenation for a fast public-key cryptosystem. *Journal of Cryptography*, 3:63–79, 1991.

[5] G.B. Agnew, R.C. Mullin, and S.A. Vanstone. On the development of a fast elliptic curve cryptosystem. In *Advances in Cryptography — EUROCRYPT '92*, pages 482–487. Springer-Verlag, 1992.

[6] G.B. Agnew, R.C. Mullin, and S.A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected areas in Communications*, 11(5):804–813, June 1993.

[7] P. Alfke and B. New. *Implementing State Machines in LCA Devices.* Xilinx, Inc., San Jose, CA. Xilinx Application Note XAPP 027.001.

[8] D. Beauregard. Efficient algorithms for implementing elliptic curve public-key schemes. Master's thesis, ECE Dept., Worcester Polytechnic Institute, Worcester, USA, May 1996.

[9] W. Gollmann. Algorithmenentwurf in der Kryptographie. Habilitation, Fakultät für Informatik, Universität Karlsruhe, Germany, August 1990.

[10] M. Gschwind and V. Salapura. A vhdl methodology for fpgas. In Will Moore and Wayne Luk, editors, *Proceedings of the 5th International Workshop on Field Programmable Logic and Applications, FPL1995*, pages 208–217, Berlin, August/September 1995. Springer-Verlag.

[11] J. Guajardo and C. Paar. Efficient algorithms for elliptic curve cryptosystems. In *Advances in Cryptography — CRYPTO '97*, pages 342–356. Springer-Verlag, 1997. LNCS 1294.

[12] G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In *Advances in Cryptology — EUROCRYPT '92*, pages 163–173, May 1992.

[13] S. Hauck, G. Borriello, S. Burns, and C. Ebeling. Montage: An fpga for synchronous and asynchronous circuits. In *2nd International Workshop on Field-Programmable Gate Arrays*, Vienna, August 1992.

[14] S. Hauck, S. Burns, G. Borriello, and C. Ebeling. An fpga for implementing asynchronous circuits. In *IEEE Design & Test of Computers*, volume 11, pages 60–69. IEEE, Fall 1994.

[15] Scott Hauck. Asynchronous design methodology: An overview. In *Proceedings of the IEEE*, volume 83, pages 69–93. IEEE, January 1995.

[16] I.S. Hsu, T.K. Truong, L.J. Deutsch, and I.S. Reed. A comparison of VLSI architecture of finite field multipliers using dual-, normal-, or standard bases. *IEEE Transactions on Computers*, 37(6):735–739, June 1988.

[17] Y. Jeong and W. Burleson. Choosing VLSI algorithms for finite field arithmetic. In *IEEE Symposium on Circuits and Systems, ISCAS 92*, 1992.

[18] D.E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[19] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[20] N. Koblitz. Constructing elliptic curve cryptosystems in characteristic 2. In *Advances in Cryptology — CRYPTO '90*, pages 156–167. Springer-Verlag, Berlin, 1991.

[21] S. Lin and D.J. Costello. *Error Control Coding: Fundamentals and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[22] E.D. Mastrovito. VLSI design for multiplication over finite fields $GF(2^m)$. In *Lecture Notes in Computer Science 357*, pages 297–309. Springer-Verlag, Berlin, March 1989.

[23] E.D. Mastrovito. *VLSI Architectures for Computation in Galois Fields.* PhD thesis, Linköping University, Dept. Electr. Eng., Linköping, Sweden, 1991.

[24] R.J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1987.

[25] A. Menezes and S. Vanstone. The implementation of elliptic curve cryptosystems. In *Advances in Cryptology — AUSCRYPT '90*, pages 2–13, January 1990.

[26] A. Menezes and S. Vanstone. Elliptic curve cryptosystems and their implementation. *Journal of Cryptography*, 6:209–224, 1993.

[27] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[28] A.J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[29] V. Miller. Uses of elliptic curves in cryptography. In *Lecture Notes in Computer Science 218: Advances in Cryptology — CRYPTO '85*, pages 417–426. Springer-Verlag, Berlin, 1986.

[30] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R.M. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics, North Holland*, 22:149–161, 1988/89.

[31] C. Paar. A parallel Galois field multiplier with low complexity based on composite fields. In *6th Joint Swedish-Russian Workshop on Information Theory*, pages 320–324, Mölle, Sweden, August 22–27 1993.

[32] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, (Engl. transl.), Institute for Experimental Mathematics, University of Essen, Essen, Germany, June 1994.

[33] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45(7):856–861, July 1996.

[34] C. Paar, P. Fleischmann, and P. Roelse. Efficient multiplier architectures for Galois fields $GF(2^{4n})$. *Preprint No. 7, Institute for Experimental Mathematics,*

University of Essen, Germany, 1996. (also submitted to the IEEE Transactions on Computers, under 2nd review).

[35] C. Paar and N. Lange. A comparative VLSI synthesis of finite field multipliers. In *3rd International Symposium on Communication Theory and its Applications*, Lake District, UK, July 10–14 1995.

[36] C. Paar and P. Soria Rodriguez. A new class of fast finite field architectures for public-key algorithms. In *Advances in Cryptography — EUROCRYPT '97*, pages 363–378. Springer-Verlag, 1997. LNCS 1233.

[37] C. Paar and M. Rosner. Comparison of arithmetic architectures for reed-solomon decoders in reconfigurable hardware. In *IEEE Symposium FPGAs for Custom Computing Machines*, 1997.

[38] A. Pincin. A new algorithm for multiplication in finite fields. *IEEE Transactions on Computers*, 38(7):1045–1049, July 1989.

[39] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. *Advances in Cryptogaphy, Crypto 95*, pages 43–56, 1995.

[40] J.H. Silverman and J. Tate. *Rational Points on Elliptic Curves.* Springer-Verlag, 1992.

[41] G.J. Simmons, editor. *Contemporary Cryptology.* IEEE Press, 1992.

[42] D. R. Stinson. *Cryptography, Theory and Practice.* CRC Press, 1995.

[43] Synopsys. *Finite State Machines - Application Note.* Synopsys, Inc., Mountain View, CA, April 1995.

[44] S.B. Wicker and V.K. Bhargava, editors. *Reed-Solomon Codes and Their Applications.* IEEE Press, 1994.

[45] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. In *Asiacrypt '96*. Springer Lecture Notes in Computer Science, 1996.

[46] Xilinx. *Gate Count Capacity Metrics for FPGAs.* Xilinx, Inc., San Jose, CA. Xilinx Application Note XAPP 059.

[47] Xilinx, Inc., San Jose, CA. *The Programmable Logic Data Book*, 1996.

[48] K. Yiu and K . Peterson. A single-chip VLSI implemenation of the discrete exponential public-key distribution system. *IBM Systems Journal*, 15(1):102–116, 1982.