Towards Efficient Hardware Implementation of Elliptic and Hyperelliptic Curve Cryptography

by

Marwa Nabil Ismail

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

Abstract

Implementation of elliptic and hyperelliptic curve cryptographic algorithms has been the focus of a great deal of recent research directed at increasing efficiency. Elliptic curve cryptography (ECC) was introduced independently by Koblitz and Miller in the 1980s. Hyperelliptic curve cryptography (HECC), a generalization of the elliptic curve case, allows a decreasing field size as the genus increases.

The work presented in this thesis examines the problems created by limited area, power, and computation time when elliptic and hyperelliptic curves are integrated into constrained devices such as wireless sensor network (WSN) and smart cards. The lack of a battery in wireless sensor network limits the processing power of these devices, but they still require security. It was widely believed that devices with such constrained resources cannot incorporate a strong HECC processor for performing cryptographic operations such as elliptic curve scalar multiplication (ECSM) or hyperelliptic curve divisor multiplication (HCDM). However, the work presented in this thesis has demonstrated the feasibility of integrating an HECC processor into such devices through the use of the proposed architecture synthesis and optimization techniques for several inversion-free algorithms.

The goal of this work is to develop a hardware implementation of binary elliptic and hyperelliptic curves. The focus is on the modeling of three factors: register allocation, operation scheduling, and storage binding. These factors were then integrated into architecture synthesis and optimization techniques in order to determine the best overall implementation suitable for constrained devices.

The main purpose of the optimization is to reduce the area and power. Through analysis of the architecture optimization techniques for both datapath and control unit synthesis, the number of registers was reduced by an average of 30%. The use of the proposed efficient explicit formula for the different algorithms also enabled a reduction in the number of read/write operations from/to the register file, which reduces the processing power consumption. As a result, an overall HECC processor requires from 1843 to 3595 slices for a Xilinix XC4VLX200 and the total computation time is limited to between 10.08 ms to 15.82 ms at a maximum frequency of 50 MHz for a varity of inversion-free coordinate systems in hyperelliptic curves. The value of the new model has been demonstrated with respect to its implementation in elliptic and hyperelliptic

curve crypogrpahic algorithms, through both synthesis and simulations.

In summary, a framework has been provided for consideration of interactions with synthesis and optimization through architecture modeling for constrained enviroments. Insights have also been presented with respect to improving the design process for cryptogrpahic algorithms through datapath and control unit analysis.

Acknowledgements

All praise be to Allah, the Creator and Sustainer of the world, for giving me the soul support to complete this thesis.

First, I would like to express my deep and sincere gratitude to my supervisor at the University of Waterloo, Professor M. Anwar Hasan, for being an outstanding advisor. His invaluable guidance, encouragement, and support from the initial to the final level enabled me to complete this thesis. It was an honour to have worked under his supervision.

I also am deeply indebted to Professor Alfred Menezes, Professor Catherine H. Gebotys, and Professor Guang Gong for serving on the thesis committee and for offering their insightful comments and invaluable suggestions. Moreover, I would like to thank Professor Amr Youssef, Concordia University for taking the time to review this work as an external examiner.

Special thanks to Dr. Jithra Adkari and Dr. Abdulaziz Alkhoraidly for providing me with valuable feedback about my work. I am very grateful to all of my collegues for creating such a great environment and for making my research so enjoyable throughout my stay at Waterloo. Thanks as well to Barbara Trotter for her assistance in proofreading my thesis. I would also like to thank the ECE administrative staff for their help, kindness, patience, and cooperation.

I warmly thank my parents for their consistent encouragement, valuable advice, and the guidance that helped make my graduate studies successful.

Special thanks go to my lovely husband, Abdelaziz Aboueleinin, for his unconditional help and understanding during these last five years while I was conducting the research for my thesis. I also would like to thank my lovely children, Mohamed, Hossam, and Habibah, who have been very patient throughout the time I was preparing and writing my thesis.

Finally, my warmest appreciation goes to the Ministry of Higher Education in Egypt, who made this thesis possible through the Bureau of Cultural and Educational Affairs of Egypt in Canada.

To all of you, I thank you very much!

To my parents, my husband, and my children

Table of Contents

List of Algorithms

List of Figures

List of Tables

List of Abbreviations

$\mathcal{R}$	***Recent*** **Coordinate**
$\mathcal{A}$	***Affine*** **Coordinate**
$\mathcal{LD}$	***López-Dahab*** **Coordinate**
$\mathcal{N}$	***New Weighted*** **Coordinate**
$\mathcal{P}$	***Projective*** **Coordinate**
ASIC	**Application Specific Integrated Circuit**
BRAM	**Block Random Access Memory**
ECC	**Elliptic curve cryptography**
ECCs	**Elliptic Curve Cryptosystems**
ECDSA	**Elliptic Curve Digital Signature Algorithm**
ECP	**Elliptic curve processor**
ECSM	**Elliptic Curve Scalar Multiplication**
FFAU	**Finite Field Arithmetic Unit**
FFs	**Flip Flops**
FP	**Forwarding Path**
FPGA	**Field Programmable Gate Array**
FSMD	**Finite State Machine with Datapath**
GCD	**Greatest Common Divisor**

HCDLP	**Hyperelliptic Curve Discrete Logarithm Problem**
HCDM	**Hyperelliptic Curve Divisor Multiplication**
HECC	**Hyperelliptic curve cryptography**
HECCs	**Hyperelliptic Curve Cryptosystems**
HECP	**Hyperelliptic curve processor**
ILP	**Instruction Level Parallelism**
LUTs	**Lookup Tables**
NIST	**National Institute of Standards and Technology**
OSFPs	**Operation Scheduling via Forwarding Paths**
PKC	**Public Key Cryptography**
PKCs	**Public Key Cryptosystem**
RAVLA	**Register Allocation via Variable Liveness Analysis**
RF	**Regsiter File**
RSA	**Rivest-Shamir-Adleman**
RTL	**Register-transfer Level**
SBERS	**Storage Binding via Efficient Register Spilling**
VHDL	**Very High Speed Integrated Circuit Hardware Description Language**
WSN	**Wireless Sensor Network**

Chapter 1

Introduction

Cryptography is the study of methods to ensure the authenticity, integrity and non-repudiation of data, and it is a fundamental concern in computer and network security. These days cryptography is incorporated into a variety of applications, including email, web browsing, banking, and electronic commerce. A major problem in the early days of cryptography was the requirement that each user have the same secret key. This key had to be exchanged using a non-encrypted method, e.g., through a personal meeting of the users or through a trusted third party. This symmetric key was then used to encrypt and decrypt messages. Public key cryptography (PKC) overcame this inefficiency through the use of asymmetrical encryption. In a PKC system, each user has two keys: one public and one private. Only the user knows the private key, while everyone is aware of the public key.

The concept of PKC was introduced in 1976 by Whitfield Diffie and Martin Hellman [26]. The security of this scheme is based on the intractability of the discrete logarithm problem in the multiplicative group of a large finite field. In 1985, Neal Koblitz [47] and Victor Miller [70] independently proposed the use of an elliptic curve group over a finite field for the implementation of public key cryptosystem (PKCs). The advantage of elliptic curves is that the discrete logarithm problem is considered much more difficult with elliptic curves than in some groups (e.g., the discrete logarithm problem in the multiplication group of finite field and also difficult that the integer factorization problem (RSA)). Elliptic curve cryptography (ECC) therefore allows smaller key sizes to be used to achieve the same level of security, which results in lower memory requirements, faster encryption and decryption, less power consumption, and lower bandwidth

requirements. Elliptic curve cryptography is consequently well suited for low-power embedded systems that require high levels of security when transferring data.

An alternative to RSA and elliptic curves is to use other curves, in particular, genus 2 curves. These cryptosystems, which have been named hyperelliptic, were proposed in 1989 [48], soon after the elliptic ones, but their deployment is far more difficult. The first problem encountered was the group law. For elliptic curves, the elements of the group are just the points of the curve. There is no group structure on the set of points of a hyperelliptic curve. Instead, in a hyperelliptic curve cryptography (HECC), the elements of the group are points of a 2-dimensional variety associated with the genus 2 curve, called the Jacobian variety. ECC and its generalization, HECC, have since been the subject of steadily increasing interest, especially with the profusion of embedded devices, such as mobile telephones, car navigation systems, and trusted computing modules. Such applications have resulted in a stronger motivation to provide algorithms for performing cryptographic operations for curve-based cryptography that require fewer computational and memory resources.

The goal of this thesis is the development of hardware oriented scalar multiplication algorithms for elliptic and hyperelliptic curve cryptography, in particular, techniques for optimizing these operations and reducing the area, power consumption, and/or computation time. The focus of the research is on curves with efficiently optimized algorithms over binary fields on an field programmable gate arrays (FPGA) based design methodology. Further important aspects of this work involved mapping the development of explicit expressions for these optimized algorithms, including the use of different coordinate systems, the implementation of explicit formulas for hyperelliptic curve divisor multiplication, and the determination of compact hardware design architecture for this operation.

1.1 Research Contribution

Hardware synthesis for cryptographic devices lies at the intersection of electrical and cryptographic engineering. Due to harsh area and power constraints associated with constrained computing devices, an increasing need has arisen for architecture synthesis and optimization solutions that are tailored to these environments.

The main contributions of this research are as follows.

- Compose an architecture synthesis methodology relying on register allocation via variable liveness analysis, operation scheduling via forwarding paths, and storage binding via efficient register spilling. Develop a modified register management for performing inversion-free operations in both ECC and HECC arithmetic.

- Appling the above-mentioned methodology, develop an efficient hardware implementation of HECC and ECC processors for inversion-free coordinates over a binary field. The design is optimized to reduce memory and register requirements.

- For genus 2 HECC, the explicit formula for *projective* ($\mathcal{P}$), *new weighted* ($\mathcal{N}$), and *recent* ($\mathcal{R}$) coordinates are optimized and a comparison is performed in order to determine appropriate inversion-free coordinates with respect to area, power and computation time of the group operations.

- To the best of the author's knowledge, this study is the first practical comparison of inversion-free coordinate hardware implementations for ECC and HECC with respect to the same level of security based on memory/register requirements, energy consumption, and computation time. This comparison also takes into account different digit sizes for finite field multipliers.

1.2 Outline

Chapter 2 introduces necessary mathematical background of elliptic and hyperelliptic curve cryptosystems. The basic definitions and properties are presented along with those of points and divisors. These explanations enable the arithmetic and the Jacobian of the ECC and the HECC to be defined. The definition of a polynomial representation of the equivalent classes is then provided, and the point and group operations based on the López-Dahab ($\mathcal{LD}$) and Cantor-Harley algorithms are introduced.

Chapter 3 provides an overview model of the stages in the hardware synthesis design flow. This chapter gives high-level understanding of various aspects of architecture synthesis and optimization design issues, including register allocation via variable liveness analysis, operation scheduling via forwarding paths, and storage binding via efficient register spilling. The chapter concludes with an explanation of datapath and control unit analysis, which offers insight into the investigation of the tradeoffs required during each step of the hardware synthesis design flow.

Chapter 4 presents our implementation of an optimized elliptic curve processor (ECP) with a special scalar multiplier and self-controlled architecture. The goal of the study in this chapter is to optimize the ECP implementation for resource restricted environments with respect to hardware usage. The register management algorithms for both conventional and optimized point multiplication are also provided. The area, power, and processing time results are also presented and compared. The parallelism is exploited on three different levels: the field arithmetic level, the point operation level, and the scalar multiplication level. López-Dahab *projective* and *mixed* coordinate formulas for genus 1 curves are targeted. The analysis has resulted in an optimized architecture for ECP.

Chapter 5 first gives a brief review of previous work with respect to HECC implementation. The improvements to the group operations for the HECC developed in the research conducted for this thesis are then presented, with an initial introduction of the methods used for minimizing register requirements in a group operation. This chapter also includes a description of optimized group operations based on the explicit formula of Cantor's and Harley's algorithms and on the *projective* and the *new weighted* coordinates introduced by Lange. The optimized HECC implementation results achieved on an FPGA are also presented in this chapter. Three different architectures have been developed for targeting area, power, and speed through the use of *projective*, *the new weighted*, and *recent* coordinates. The HECC processor is described, and the methodology for different design options is outlined. This chapter ends with a comparison of the results related to HECC implementations using FPGAs.

Chapter 6 introduces the theoretical comparison of a variety of different architecture options for ECC and their HECC equivalents on the proposed processor. The focus is on the underlying area, power and processing time. The theoretical comparison matrices are validated using FPGA implementations. This work finishes with a comparison of the area, power, and computation time of ECC for a recommended finite field library.

The conclusions arising from this work and some suggestions for further research are summarized in Chapter 7. The Appendix contains additional information such as the VHDL codes used for the hardware designs and some special implementation issues.

Chapter 2

Preliminary Background

This chapter gives background related to finite field operations and elliptic curve cryptography. It mainly focuses on field $\mathbb{F}_{2^{163}}$ which is one of the five binary fields recommended by the National Institute of Standards and Technology (NIST) for Elliptic Curve Digital Signature Algorithm (ECDSA) application [75]. This chapter also provides an elementary introduction to the Jacobian of hyperelliptic curves over finite fields of even characteristic, with attention being given only to definitions and algorithms that are relevant for this work. References [46, 54, 37, 7] offer additional details about these topics.

2.1 Binary Field Arithmetic

The finite field $\mathbb{F}_2$ has two elements; 0 and 1. The addition and multiplication are performed modulo 2 as in two's complement arithmetic. $\mathbb{F}_{2^m}$ is the extension field of $\mathbb{F}_2$ and has 2^m elements. Each of these elements is represented as a polynomial of degree less than or equal to $m-1$ with coefficients coming from the ground field $\mathbb{F}_2$. For such a representation, addition is bit-independent and straightforward. However, multiplication and squaring involve polynomial multiplication and squaring modulo an irreducible polynomial of degree m. Hence, the design of efficient architectures to perform these arithmetic operations is of great practical concern. This section summarizes the arithmetic and architectures for field operations. We will concentrate on fields $\mathbb{F}_{2^m}$, because fields of characteristic 2 are best suited for hardware architectures. An element $A \in \mathbb{F}_{2^m}$ will be represented as a polynomial $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $a_i \in \mathbb{F}_2$. The irreducible polynomial will be denoted as $F(x)$. The list below provides a short description of each imple-

Algorithm 2.1 Bit serial multiplication (low to high bits)

```
Input:  An irreducible polynomial $F(x)$ of degree $m$, two elements $A(x), B(x) \in \mathbb{F}_{2^m}$.
Output:  $C(x) = A(x)B(x) \bmod F(x)$.
1: $C(x) = 0$
2: for $i = 0$ to $m-1$ do
3:     $C(x) = b_i A(x) + C(x)$
4:     $A(x) = A(x) \cdot x \bmod F(x)$
5: end for
6: return $C(x)$
```

mentation along with references pointing to more detailed information for field multiplication.

2.1.1 Field Addition Operation

The addition of two elements, A and B, in a binary field $\mathbb{F}_{2^m}$ is performed by a bit wise XORing with no carry propagation $(C(x) \equiv \sum_{i=0}^{m-1} c_i x^i) = (A(x) \equiv \sum_{i=0}^{m-1} a_i x^i) \oplus (B(x) \equiv \sum_{i=0}^{m-1} b_i x^i)$. This means:

$$C(x) = \sum_{i=0}^{m-1} c_i x^i = A(x) + B(x) = \sum_{i=0}^{m-1} ((a_i + b_i) \,\mathbf{mod}\, 2)\, x^i$$

where $c_i, a_i, b_i \in \mathbb{F}_2$. It should be noted that in characteristic 2, subtraction of two field elements is the same as addition because each element is its own additive inverse.

2.1.2 Field Multiplication Operation

A number of polynomial basis bit-serial, digit-serial, and bit-parallel finite field multipliers have been proposed. The bit-parallel type multipliers have small critical path delay and high throughput, and are best suited for applications requiring high speed. If two arbitrary field elements A and $B \in \mathbb{F}_{2^m}$ are expressed as $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$, then the product of A and B can be expressed as,

$$C(x) = A(x)B(x)\,\mathbf{mod}\,F(x) = A(x)\left(\sum_{i=0}^{m-1} b_i x^i\right)\,\mathbf{mod}\,F(x) = \left(\sum_{i=0}^{m-1} b_i A(x) x^i\right)\,\mathbf{mod}\,F(x)$$

Algorithm 2.2 Digit-serial multiplication (low to high bits)

```
Input:  $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{n_G-1} B_i(x)\, x^{Gi}$, where $B_i(x)$ as in Equation (2.2)
Output:  $C(x) = A(x)\, B(x) = \sum_{i=0}^{m-1} c_i x^i$,  where $c_i \in \mathbb{F}_2$
1: $C(x) = 0$
2: for $i = 0$ to $n_G - 1$ do
3:     $C(x) = B_i(x)\, A(x) + C(x)$
4:     $A(x) = A(x) \cdot x^G \bmod F(x)$
5: end for
6: return $C(x)$‘
```

where $F(x)$ is an irreducible binary polynomial of degree m and defines the field $\mathbb{F}_{2^m}$. Therefore,

$$C(x) = (b_0 A(x) + b_1 A(x)x + b_2 A(x)x^2 + ... + b_{m-1} A(x)x^{m-1})\,\mathbf{mod}\; F(x). \qquad (2.1)$$

Algorithm 2.1 shows a well-known procedure for implementing Equation (2.1). The reader is referred to [37] for a diagram of this kind of bit serial multiplier. Other multipliers (e.g. digit serial and bit parallel) require fewer clock cycles but more space. The bit serial multiplier provides the most area efficient method of implementing hardware. The computation time, area, and power requirements for the developed implementation of the bit serial multiplier are given in Table 2.1 for the case $m = 163$.

Bit serial multipliers are very area efficient, but they are quite slow. On the other hand, fully bit parallel multipliers require too much area. For the purpose of this work, we consider digit serial multipliers. Such multipliers process multiple bits of the input word in one clock cycle. The digit-size can be varied in order to achieve a desired level of tradeoff between area, power, and speed. In this work, the digit-serial finite field multiplier proposed by [86] was used as shown in Algorithm (2.2). This system is called digit-serial/parallel, which reflect the fact that the multiplicand bits are processed in parallel while the multiplier bits are processed one digit at a time. The design has two major computations: i) partial product generation and accumulation as seen in Step 3 of Algorithm (2.2), and ii) mod $F(x) = x^m + f_n x^n + \sum_{i=0}^{n-1} f_i x^i$ degree reduction operations as seen in Step 4 of Algorithm (2.2). If G is assumed to be the digit size; n_G denotes the total number

Table 2.1: FPGA Results for Virtex 4: XC4vlx200-11ff1513

Design	Total		Area	F_{max}	Time	Power (mW)	
	$\#FFs$	$\#LUTs$	$slices$	MHz	ns	Dynamic	Leakage
Bit Multiplier_163	500	513	263	292.895	620.02	361.48	1359.96
Digit Multiplier_163 (G = 2)	762	857	440	287.455	317.92	383.08	1370.99
Digit Multiplier_163 (G = 4)	1409	1486	765	285.225	154.26	384.91	1371.16
Digit Multiplier_163 (G = 8)	2788	2913	1500	266.736	80.87	320.05	1365.26
Digit Multiplier_163 (G = 16)	5674	6020	3098	244.899	40.18	323.62	1365.58
Field_Squarer_163	-	165	88	168.63	5.93	1.22	1354.77

of digits with $n_G = \lceil m/G \rceil$; and $A(x) = \sum_{i=0}^{m-1} A_i(x)x^i$, $B(x) = \sum_{i=0}^{n_G-1} B_i(x)x^{Gi}$, where

$$B_i(x) = \begin{cases} \sum_{j=0}^{G-1} B_{Gi+j}(x)x^j, & 0 \leq i \leq G-2 \\ \sum_{j=0}^{m-1-G(n_G-1)} B_{Gi+j}(x)x^j, & i = G-1 \end{cases} \tag{2.2}$$

$$C(x) = A(x)\,B(x)\,\mathbf{mod}\,F(x) = A(x)\sum_{i=0}^{G-1} B_i(x)x^{Gi}\mathbf{mod}\,F(x) \tag{2.3}$$

The following equation is thus derived for the least significant digit-serial (LSD) multiplier scheme:

$$\begin{aligned} C(x) = (B_0(x)\,A(x) + B_1(x)\,(A(x) \cdot\, x^G\mathbf{mod}\,F(x) \\ + B_2(x)\,(A(x)x^G \cdot\, x^G\mathbf{mod}\,F(x)) + \cdots \\ + B_{n_G-1}(A(x)x^{G(n_G-2)} \cdot\, x^G\mathbf{mod}\,F(x))) \\ \mathbf{mod}\,F(x) \end{aligned} \tag{2.4}$$

The multiplication of two field elements can then be expressed as follows:

$$(A(x)\sum_{i=0}^{n_G-1} B_i(x)x^{Gi})\ \mathbf{mod}F(x) \equiv (\sum_{i=0}^{n_G-1} B_i(x)(A(x)x^{Gi}\ \mathbf{mod}F(x)))\ \mathbf{mod}F(x)$$

For digit multipliers with digit size G, when $G \leq m-n$, Step 3 and Step 4 in Algorithm (2.2) can be reduced to degree less than m in one step [86]. In this work a different number of the digit-size is investigated in order to reduce the computation time as well as the area and power consumption.

2.1.3 Field Squaring Operation

While squaring is a special case of general multiplication and can be performed by a multiplier, performance can be significantly improved through the optimization of the architecture. Squaring a field element in $\mathbb{F}_{2^m}$ represented via a polynomial basis, $\{1, x, x^2, \ldots, x^{m-1}\}$, is ruled by the following equation.

$$A^2(x) = (\sum_{i=0}^{m-1} A_i x^i)^2 = \sum_{i=0}^{m-1} A_i x^{2i}\ \mathbf{mod}\,F(x) \tag{2.5}$$

The square of an element $A(x)$ in $\mathbb{F}_{2^{163}}$ represented by

$$A^2(x) = \underbrace{(a_{162}x^{160} + \cdots + a_{83}x^2 + a_{82})x^{164} \, \mathbf{mod} \; F(x)}_{A_H} + \underbrace{(a_{81}x^{162} + \cdots + a_1x^2 + a_0)}_{A_L} \quad (2.6)$$

involves three mathematical steps: (i) expand the A_L part with interleaved $0's$; (ii) reduce the A_H part with the reduction polynomial $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$; and (iii) add the two parts: A_H , A_L. However, for hardware implementations, the use of Equation (2.6) enables these three steps to be combined in four level XOR gates if the reduction polynomial has a smaller second-highest degree, which is the case here. The squaring can hence be efficiently implemented in order to generate the result in one clock cycle without huge area requirements [39]. The implementation and the area costs are shown in Table 2.1. Using a fixed reduction polynomial $F(x)$, squaring can be performed within one clock cycle using a combinatorial logic implementation. The squaring requires at most $(n-1) + (m-1)$ gates, where n represents the number of non-zero coefficients of the reduction field polynomial.

2.2 Elliptic Curve Arithmetic

An ECC-based cryptosystem is considered to be one of the best candidates for light-weight applications, such as mobile devices, due to its small key size and efficient computation [28]. Cryptographic mechanisms based on elliptic curves depend on the arithmetic of points on the curve. Elliptic curve arithmetic is defined in terms of the underlying field operations, which are described in Section 2.1. A non-supersingular elliptic curve over $\mathbb{F}_{2^m}$ can be defined as follows:

$$E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + ax^2 + b \, , \; b \neq 0 \quad (2.7)$$

In its simplest form an elliptic curve point $P \in E$ is defined as a pair of elements (x, y) of $\mathbb{F}_{2^m}$ that satisfying Equation (2.7). The points on E form a commutative finite group under the point addition operation. For the work presented in this thesis, a NIST-recommended random curve [75] with a parameter $b \neq 0$ was employed. The special point $\mathcal{O}$, known as the point at infinity, is the additive identity of the group. The addition of the two points on E is performed using the well-known "chord and tangent" process [14]. The underlying opera-

Algorithm 2.3 Left-to-right binary method for scalar multiplication

```
Input: $P \in E(\mathbb{F}_{2^m})$, $k = \sum_{i=0}^{l} k_i 2^i$ where $k_{l-1} = 1$
Output: $kP \in E(\mathbb{F}_{2^m})$
1: $Q \leftarrow P$
2: for $l-2$ down to 0 do
3:     $Q \leftarrow 2Q$
4:     if $k_i = 1$ then
5:         $Q \leftarrow Q + P$
6:     end if
7: end for
8: return $Q = kP$
```

tions used to perform the point addition are $\mathbb{F}_{2^m}$ arithmetic operations. Point doubling is a special case of point addition, and scalar multiplication kP is the addition of k copies of point P i.e., $Q = kP = \underbrace{P + P + P + \cdots + P}_{k\ copies}$. For a large value of k, the scalar multiplication can be performed using repeated point additions and doubling, as described in Algorithm 2.3.

2.2.1 Example on an Elliptic Curve over $\mathbb{F}_{2^{163}}$

In this subsection, an example of a point on an elliptic curve with almost-prime group order [37] is given. The underlying field is $\mathbb{F}_{2^{163}}$ with reduction polynomial $F(z) = z^{163} + z^7 + z^6 + z^3 + 1$. The example is the random elliptic curve $E(\mathbb{F}_{2^{163}})$ of genus 1 defined by

$$y^2 + xy = x^3 + ax^2 + b\ , \quad a = 1$$

where

$$b = \texttt{000000020A601907B8C953CA1481EB10512F78744A3205FD}$$

$$x = \texttt{00000003F0EBA16286A2D57EA0991168D4994637E8343E36}$$

$$y = \texttt{00000000D51FBC6C71A0094FA2CDD545B11C5C0C797324F1}$$

are given in hexadecimal. The order of the base point P over $\mathbb{F}_{2^{163}}$ is

$$n = \underbrace{\texttt{000000040000000000000000000292FE77E70C12A4234C33}}_{l}$$

where the factor l is prime. In cryptographic applications with this group, the multiplier for scalar multiplication should be an integer k, where $1 \leq k \leq l-1$. In this case l is 163 bits in length, so k should also be (at most) 163 bits.

2.2.2 Point Representations

Elliptic curve points can be represented using various coordinate systems such as affine or projective representations. For each such system, the speed of additions and doubling is different. We consider projective presented by López-Dahab ($\mathcal{LD}$) [54] and give the appropriate formulas and the number of operations. A projective coordinate elliptic curve point $P = (X, Y, Z)$ consists of three elements of $\mathbb{F}_{2^m}$. To convert the affine point (x, y) to projective coordinates, Z is simply set to 1, i.e., $(x, y, 1)$. Projective coordinates are generally used for internal computations, but the resultant projective point is converted to its affine form before being transmitted. Different types of projective coordinates vary with respect to how the projective point maps to an affine point. As a means of avoiding an expensive field inversion operation, it is convenient to work with the projective coordinates presented by $\mathcal{LD}$ [54] because the latter is efficient and has been used previously in hardware realization.

A López-Dahab [54] projective point $P = (X, Y, Z)$ maps to the affine point $P = (X/Z, Y/Z^2)$. An effort has been made to minimize the number of finite field multiplication operations if *mixed* coordinates are used. For convenience, explicit formulas have been given below for computing point addition $P + Q = (X_3, Y_3, Z_3)$ in $\mathcal{LD}$ coordinates of $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2, Z_2)$.

$$\begin{aligned} X_3 =& (A * (D + B^2) + B * (A^2 + C)), \\ Z_3 =& (A + B^2) * Z_1 * Z_2, \\ Y_3 =& (A * H + F * C) * F + (H + Z_3) * X_3, \end{aligned} \tag{2.8}$$

where

$$\begin{aligned} A =& X_1 * Z_2, \quad B = X_2 * Z_1, \quad C = Y_1 * Z_2^2, \\ D =& Y_2 * Z_1^2, \quad E = A + B, \quad F = A^2 + B^2, \\ G =& C + D, \quad H = G * E. \end{aligned} \tag{2.9}$$

The projective form of the point doubling is $2(X_1, Y_1, Z_1) = (X_3, Y_3, Z_3)$.

$$\begin{aligned} Z_3 =& A^2, \\ X_3 =& C^2 + D + Z_3, \\ Y_3 =& (Z_3 + D) * X_3 + B^2 * Z_3, \end{aligned} \tag{2.10}$$

where

$$A = X_1 * Z_1, B = X_1^2 \quad C = B + Y_1, D = A * C.$$

Algorithm 2.3 is the left-to-right version of the basic repeated double-and-add method for point multiplication [37]. For scalar multiplication using *mixed* coordinates, Q is stored in *projective* coordinates (X_1, Y_1, Z_1), while P is stored in *affine* coordinates $(\mathcal{A})$ $(X_2, Y_2, 1)$. The point addition operation can be performed using the following formulas:

$$\begin{aligned} X_3 =& A^2 + D + E, \\ Y_3 =& (E + Z_3) * F + G, \\ Z_3 =& C^2, \end{aligned} \tag{2.11}$$

where

$$\begin{aligned} A =& Y_2 * Z_1^2 + Y_1, & B =& X_2 * Z_1 + X_1, & C =& Z_1 * B, \\ D =& B^2 * (C + aZ_1^2), & E =& A * C, & F =& X_3 + X_2 * Z_3, \\ G =& (X_2 + Y_2) * Z_3^2. \end{aligned}$$

The point addition and doubling operations iterate through the binary expansion of k [37]. On each iteration, a point doubling is performed. If $k_i = 1$, then a point addition is performed. In general, a binary expansion of k will have approximately m bits. On average, half of these bits will be equal to one. The average cost of implementing a scalar multiplication using Algorithm 2.3 is therefore given as $N_{binary} = mN_{double} + (m/2)N_{add}$, where N_{double} and N_{add} are the number of cycles for each point doubling and addition operation, respectively.

2.2.3 Point Multiplication Costs

Estimates for point multiplication costs are presented in terms of curve operations (point additions and point doubling) and in terms of the corresponding field operations (multiplications MUL, squaring SQR, addition ADD, and inversion INV). Point addition and doubling in projective coordinates cost $14\,MUL{+}4\,SQR{+}9\,ADD$ and $5\,MUL{+}4\,SQR{+}5\,ADD$ operation counts, respectively. The cost of conversion back to affine coordinates is $2\,MUL + 1\,INV + 1\,SQR$. The special case of $a = 0$ *or* 1 provides a saving of $1\,MUL$ for both point addition and doubling. For the field $\mathbb{F}_{2^{163}}$, the inversion can be implemented in $9MUL{+}162SQR$ operations. In this work the NIST-recommended elliptic curve over $\mathbb{F}_{2^{163}}$ is used

Table 2.2: Scalar multiplication (kP) time in cycles (for $k = (k_{162}, \ldots, k_1, k_0)_2$)

Coordinate System	N_{add}	N_{double}	$(kP)_{Average}$
Affine ($\mathcal{A}$)	1,793	1,793	437,492
$\mathcal{LD}$ *projective*	2,132	661	280,435
$\mathcal{LD}$ *mixed*	1,318	661	214,501

so that $a = 1$. When mixed coordinates are used, the operation counts for elliptic curve point addition and point doubling are $8\,MUL + 5\,SQR + 9\,ADD$ and $4\,MUL + 4\,SQR + 5\,ADD$, respectively, in the López-Dahab coordinate systems.

The number of cycles required for performing scalar multiplication is illustrated in Table 2.2. The corresponding number of cycles for performing point multiplication can be used to estimate the scalar multiplication times which are given in the right most column of the table.

2.3 Hyperelliptic Curve Arithmetic

The main benefit of HECC is that it offers security equivalent to that of ECC for much smaller parameter sizes. This advantage results in smaller datapaths, less memory usage, and lower power consumption [79]. Integrating a public key cryptosystem (PKCs) into a constrained device is a challenge due to the limitations in area and power. In the past, it was widely believed that devices with such constrained resources cannot carry out strong cryptographic operations such as hyperelliptic curve divisor multiplication (HCDM). However, the feasibility of integrating PKCs into such devices has recently been demonstrated by the success of several implementations [36, 13, 94, 3, 11, 49, 35].

This section provides an elementary introduction to the mathematical background needed for the application of hyperelliptic curves. Basic definitions and properties of HECC are given as well as an introductory treatment of divisors. With these definitions, the Jacobian of hyperelliptic curves over finite fields of even characteristic can be defined. In order to work efficiently with divisors, Mumford's representation is introduced [74]. Finally, we present the algorithms for addition and doubling of two elements on the Jacobian of HECC with attention being given only to definitions and algorithms that are relevant for this work. Additional details are available in [48, 46].

The idea behind using hyperelliptic curves in public key cryptography is that

groups formed from hyperelliptic curves are suitable for discrete logarithm cryptosystems. This idea was first introduced in 1989 by Neal Koblitz [48]. Hyperelliptic curves are a special class of algebraic curves and can be viewed as a generalization of elliptic curves because there are hyperelliptic curves of every genus $g \geq 1$, and a hyperelliptic curve of genus $g = 1$ is an elliptic curve. This section concentrates on genus 2 (hyperelliptic) curves over fields of characteristic 2 and in particular, provides details about divisor doubling and divisor addition formulas for different types of even characteristic curves. Choosing curves defined over $\mathbb{F}_{2^m}$ allows very efficient divisor multiplication, as shown in the literature related to even characteristic curves [59, 61].

2.3.1 Basic Definitions and Properties

Definition 1 ***A field is a set $\mathbb{F}$ with multiplication and addition operations that satisfy the familiar rules – associativity and commutativity of both addition and multiplication, the distributive law, existence of an additive identity 0 and a multiplicative identity 1, additive inverses, and multiplicative inverses for everything except 0.***

Arithmetic operations in the finite field $\mathbb{F}_{2^m}$ are used extensively in hyperelliptic curve cryptosystems. A polynomial basis is one of the common bases used as a representation of field elements of $\mathbb{F}_{2^m}$. The literature includes a number of proposed hardware structures for an $\mathbb{F}_{2^m}$ HECC processor using a polynomial basis [94, 93].

Definition 2 ***If a field $\mathbb{F}$ has the property that every polynomial with coefficients in $\mathbb{F}$ factors completely into linear factors, then $\overline{\mathbb{F}}$ can be said to be algebraically closed. For example, the algebraic closure of the field of real numbers is the field of complex numbers.***

If $\mathbb{F}$ is a field and $\overline{\mathbb{F}}$ is the algebraic closure of $\mathbb{F}$, a hyperelliptic curve C of genus g over $\mathbb{F}$ $(g \geq 1)$ is given by an equation of the form

$$C \,:\, y^2 + h(x)y = f(x) \qquad in \qquad \mathbb{F}[x,\, y]\,, \tag{2.12}$$

where $h(x) \in \mathbb{F}[x]$ is a polynomial of degree at most g $(deg(h) \leq g)$, $f(x) \in \mathbb{F}[x]$ is a monic polynomial of degree $2g+1$ $(deg(f) = 2g+1)$, and there are no singular points.

Definition 3 *A* singular point *on C is a solution $(x, y) \in \overline{\mathbb{F}} \times \overline{\mathbb{F}}$ that simultaneously satisfies Equation (2.12) and the following partial differential equation:*

$$2y + h(x) = 0, \quad h'(x)y - f'(x) = 0. \tag{2.13}$$

Definition 4 *A curve is said to be* non-singular *if there are no pairs $(x, y) \in \overline{\mathbb{F}} \times \overline{\mathbb{F}}$ that simultaneously satisfy the equation of the curve C and the partial differential equations $2y + h(x) = 0$ and $h'(x)y - f'(x) = 0$.*

Definition 5 *A divisor D is a formal sum of points on a hyperelliptic curve C. The Jacobian J_C is the group of degree 0 divisors modulo the principal divisors.*

Definition 6 *A principal divisor is the divisor of a rational function. Let $f \in \bar{\mathbb{F}}(C)$ be a rational function on C.*

1. *The divisor of f is defined as*

$$\boldsymbol{div}(f) = \sum_{P \in C} \boldsymbol{ord}_P(f) P.$$

2. *A divisor D is called principal if $D = \boldsymbol{div}(f)$ for some rational function $f \in \bar{\mathbb{F}}(C)$.*

3. *The set of principal divisors on C is denoted by $\boldsymbol{Princ}(C)$.*

In practice, the Mumford representation [74] is used for elements of $\mathbb{J}_C$: each divisor is represented by a pair of polynomials (u, v) such that u is a monic polynomial of degree at most 2, deg v <deg $u \leq 2$, and $u \mid v^2 + vh - f$; these types of divisors are called reduced. This representation provides a very compact definition of divisor classes and also allows to easily use divisor classes in implementations since only two lists of coefficients (of the two polynomials u and v) of length at most g have to be stored in a computer [62].

2.3.2 Genus 2 Hyperelliptic Curve over $\mathbb{F}_{2^m}$

The *non-singular* curve $C : y^2 + h(x)y = f(x),\ h,\ f \in \mathbb{F}_{2^m}$, deg$(f) = 5$, deg$(h) \leq 2$, is called *a Hyperelliptic curve of genus 2 over* $\mathbb{F}_{2^m}$, where $h(x) = h_2x^2 + h_1x + h_0$ and $f(x) = x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$ are polynomials defined over $\mathbb{F}_{2^m}$. Genus 2 curves in characteristic 2 can be divided into three types depending on the polynomial h : Type 1 if deg $h = 2$, Type 2 if deg $h = 1$, and Type 3 if deg $h = 0$. A genus 2 curve of Type 2 defined over $\mathbb{F}_{2^m}$ by an equation of the form $y^2 + h(x)y = f(x)$ is isomorphic to a curve defined by an equation of the form $y^2+xy = x^5+f_3x^3+\varepsilon x^2+f_0$, where ε is in $\mathbb{F}_2$ if m is odd. The most common case in cryptographic applications is that m should be prime in order to resist potential Weil descent attacks [22].

The points on the curve C do not form a group, so the divisor class group of C (also called the Jacobian) is used instead. The elements of this group are represented using Mumford's representation as described in Section 2.3.1.

2.3.3 Example of a Genus 2 Hyperelliptic Curve over $\mathbb{F}_{2^{83}}$

In this subsection, an example is provided of the Jacobian of an hyperelliptic curve with almost-prime group order [24]. The correctness of this example can be easily checked through multiplying a random divisor by the given group order and then verifying that the result is principal [24]. The underlying field is $\mathbb{F}_{2^{83}}$ with reduction polynomial $F(z) = z^{83}+z^7+z^4+z^2+1$. The example is the random hyperelliptic curve C of genus 2 defined by

$$y^2 + (\sum_{i=0}^{2} h_i x^i)y = x^5 + \sum_{i=0}^{4} f_i x^i,$$

where

$$\begin{aligned}
h_0 &= \texttt{7FF29B08993336B479CD2} & \qquad h_1 &= \texttt{32C101713C722F8FB5BC9}\\
h_2 &= \texttt{553E16B6A3BC6B2432CA8} & & \\
f_0 &= \texttt{7AD44882C02B9743CD58B} & \qquad f_1 &= \texttt{327254FA330B44958262A}\\
f_2 &= \texttt{204AB23E12828D061AF04} & \qquad f_3 &= \texttt{1C827250FFDEFF93B43BE}\\
f_4 &= \texttt{13D80106C0E5571DFE139} & &
\end{aligned}$$

are given in hexadecimal. The group order of the Jacobian $\mathbb{J}_C$ over $\mathbb{F}_{2^{83}}$ is

$$\#\mathbb{J}_C = 2 \cdot \underbrace{46768052394566313810931349196246034325781246483037}_{l},$$

where the last factor l is prime. In cryptographic applications with this group, the multiplier for divisor multiplication should be an integer k, where $1 \leq k \leq l-1$.

In this case l is 165 bits in length, so k should also be (at most) 165 bits.

2.3.4 Divisor Class Representations

A divisor $D = \sum_{P \in C} k_i P_i,\ k_i \in \mathbb{Z}$, is a formal sum of points on a hyperelliptic curve C. The degree of D, denoted deg D, is the integer $\sum_{P \in C} m_i$. The set of all divisors form an additive group denoted by $\mathbb{D}(C)$. The set of divisors of degree zero are denoted as $\mathbb{D}^0 \subset \mathbb{D}(C)$. A divisor $D \in \mathbb{D}^0$ is called a principal divisor if its points are the zeros and poles of a rational function on C. The set of all principal divisors is denoted by $\mathbb{P}$. The set $\mathbb{P}$ is a subgroup of $\mathbb{D}^0$. If $D_1, D_2 \in \mathbb{D}^0$ then one writes $D_1 \sim D_2$ if $D_1 - D_2 \in \mathbb{P}$; D_1 and D_2 are said to be equivalent divisors. For each divisor $D \in \mathbb{D}^0$ there exists a semi-reduced divisor $D_1 \in \mathbb{D}^0$ such that $D_1 \sim D_2$. The Jacobian of C can then be defined as the quotient group $\mathbb{J} = \mathbb{D}^0/\mathbb{P}$. The Jacobian over $\mathbb{F}$, denoted by $\mathbb{J}_C(\mathbb{F})$, consists of divisors $D = \sum k_i P_i$ for which $D^\sigma = \sum k_i P_i^\sigma$ is equal to D for all automorphisms σ of $\bar{\mathbb{F}}$ over $\mathbb{F}$. It should be noted that a divisor being defined over $\mathbb{F}$ does not mean that each P_i^σ is equal to P_i; σ may permute the points [22].

In [16], Cantor shows that each element of the Jacobian can be represented in the form $D = \sum_{i=1}^{r} P_i - r \cdot \infty$ such that for all $i \neq j$, P_i and P_j are not symmetric points (A point $P = (x, y)$ is symmetric if $P = (x, -y - h(x))$). Such a divisor is called a semi-reduced divisor. Cantor concludes from the Riemann-Roch Theorem that each element of the Jacobian can be represented uniquely by such a divisor, subject to the additional constraint $r \leq g$. Such divisors are referred to as reduced divisors. Finally, Cantor shows that the divisors of the Jacobian can be represented as a pair of polynomials u and v, where u is monic, deg $v(v) <$ deg $u(u) \leq g$, and u divides $v^2 + h(u)v - f(u)$, where the coefficients of u and v are elements of $\mathbb{F}$. A divisor D represented by such polynomials is denoted as $D(u, v)$.

Hyperelliptic curve points can be represented using different coordinate systems. For each such system, the speed of divisor addition and divisor doubling is different. In this work, *projective*, *new weighted*, and *recent* coordinate representations are considered, and the appropriate inversion-free formulas and the number of finite field operations are given. In hardware applications, an inversion-free formula is preferred because an inverter is very expensive compared to other components. In addition, the $INV/MUL-$ratio depends on the platform. A *projective* system introduces an additional coordinate called Z, as for

Table 2.3: Different coordinates *projective* ($\mathcal{P}$), *new weighted* ($\mathcal{N}$), and *recent* ($\mathcal{R}$) with approximate complexity, in even characteristic, for divisor doubling with $g = 2$

Operation	Complexity ($h_2 \neq 0$)	Complexity ($h_2 = 0$)
$2\mathcal{N} = \mathcal{P}$	$37\,MUL + 6\,SQR + 25\,ADD$	$33\,MUL + 6\,SQR + 28\,ADD$
$2\mathcal{P} = \mathcal{P}$	$38\,MUL + 7\,SQR + 39 ADD$	$29\,MUL + 7\,SQR + 23\,ADD$
$2\mathcal{R} = \mathcal{P}$	$29\,MUL + 8\,SQR + 21\,ADD$	$24\,MUL + 7\,SQR + 18\,ADD$
$2\mathcal{N} = \mathcal{N}$	$37\,MUL + 6\,SQR + 36\,ADD$	$26\,MUL + 6\,SQR + 27\,ADD$
$2\mathcal{P} = \mathcal{N}$	$35\,MUL + 6\,SQR + 27\,ADD$	$32\,MUL + 6\,SQR + 23\,ADD$
$2\mathcal{R} = \mathcal{N}$	$39\,MUL + 6\,SQR + 33\,ADD$	$35\,MUL + 5\,SQR + 29\,ADD$
$2\mathcal{N} = \mathcal{R}$	$35\,MUL + 6\,SQR + 31\,ADD$	$31\,MUL + 7\,SQR + 23\,ADD$
$2\mathcal{P} = \mathcal{R}$	$37\,MUL + 6\,SQR + 38\,ADD$	$29\,MUL + 5\,SQR + 21\,ADD$
$2\mathcal{R} = \mathcal{R}$	$37\,MUL + 5\,SQR + 36\,ADD$	$23\,MUL + 8\,SQR + 9\,ADD$
$2\mathcal{A} = \mathcal{N}$	$20\,MUL + 5\,SQR + 24\,ADD$	$19\,MUL + 6\,SQR + 20\,ADD$
$2\mathcal{A} = \mathcal{P}$	$24\,MUL + 5\,SQR + 18\,ADD$	$21\,MUL + 6\,SQR + 19\,ADD$
$2\mathcal{A} = \mathcal{R}$	$30\,MUL + 8\,SQR + 32\,ADD$	$23\,MUL + 7\,SQR + 29\,ADD$
$2\mathcal{A} = \mathcal{A}$	$1\,INV + 20\,MUL + 5\,SQR + 34\,ADD$	$1\,INV + 13\,MUL + 5\,SQR + 24\,ADD$

an elliptic curve, and lets the quintuple $[U_1,\, U_0,\, V_1,\, V_0,\, Z]$ corresponds to the divisor class $[x^2{+}U_1/Z\,x{+}U_0/Z,\; V_1/Z\,x{+}V_0/Z]$ in a Mumford representation. Projective coordinates are generally used for internal computations, but the resultant projective needs one inversion and four multiplications to be converted to its affine form before being transmitted. This idea was first proposed for genus 2 curves in [97] and then largely improved and generalized by [61].

***Recent* coordinates, $[U_1, U_0, V_1, V_0,\, Z\,,\, z]$ as defined in section 14.5 in [22] corresponds to $[x^2 + U_1/Z\,x + U_0/Z,\; V_1/Z^2\,x + V_0/Z^2]$ and $z = Z^2$. If *new weighted* coordinates are used, one more entry than for projective coordinates is needed. Thus, a divisor $[U_1,\, U_0,\, V_1,\, V_0,\, Z_1,\, Z_2]$ corresponds to the affine divisor $[x^2 + U_1/Z_1^2\,x + U_0/Z_1^2,\; V_1/(Z_1^3 Z_2)x + V_0/(Z_1^3 Z_2)]$ so that the divisor doubling and divisor addition operations can be efficiently performed. The divisor operations iterates through the binary expansion of k [37]. During each iteration, a divisor doubling is performed. If $k_i = 1$, then a divisor addition is performed. If C is a genus 2 hyperelliptic curve over $\mathbb{F}_{2^m}$ of almost-prime order, then scalars k in cryptographic applications are generally $2m$ bits in length. On average, half of these bits are equal to one. Therefore, the cost of implementing a divisor multiplication using Algorithm 2.4 is given as $(2m)\,D_{DBL} + (m)\,D_{ADD}$, where D_{DBL} and D_{ADD} are the number of cycles for each divisor doubling and divisor addition operation, respectively.**

Table 2.4: Different coordinates *projective* ($\mathcal{P}$), *new weighted* ($\mathcal{N}$), and *recent* ($\mathcal{R}$) with approximate complexity, in even characteristic, for divisor addition with $g = 2$

Operation	Complexity ($h_2 \neq 0$)	Complexity ($h_2 = 0$)
$\mathcal{N} + \mathcal{P} = \mathcal{P}$	$48\,MUL + 4\,SQR + 38\,ADD$	$46\,MUL + 4\,SQR + 35\,ADD$
$\mathcal{P} + \mathcal{P} = \mathcal{P}$	$49\,MUL + 4\,SQR + 41\,ADD$	$45\,MUL + 5\,SQR + 30\,ADD$
$\mathcal{R} + \mathcal{P} = \mathcal{P}$	$38\,MUL + 7\,SQR + 32\,ADD$	$35\,MUL + 8\,SQR + 29\,ADD$
$\mathcal{N} + \mathcal{N} = \mathcal{N}$	$48\,MUL + 4\,SQR + 35\,ADD$	$45\,MUL + 5\,SQR + 31\,ADD$
$\mathcal{P} + \mathcal{N} = \mathcal{N}$	$47\,MUL + 4\,SQR + 39\,ADD$	$45\,MUL + 4\,SQR + 38\,ADD$
$\mathcal{R} + \mathcal{N} = \mathcal{N}$	$39\,MUL + 8\,SQR + 38\,ADD$	$36\,MUL + 6\,SQR + 35\,ADD$
$\mathcal{R} + \mathcal{R} = \mathcal{R}$	$45\,MUL + 3\,SQR + 32\,ADD$	$50\,MUL + 8\,SQR + 29\,ADD$
$\mathcal{N} + \mathcal{R} = \mathcal{R}$	$37\,MUL + 6\,SQR + 29\,ADD$	$35\,MUL + 7\,SQR + 30\,ADD$
$\mathcal{P} + \mathcal{R} = \mathcal{R}$	$34\,MUL + 8\,SQR + 31\,ADD$	$32\,MUL + 5\,SQR + 28\,ADD$
$\mathcal{A} + \mathcal{N} = \mathcal{N}$	$35\,MUL + 5\,SQR + 29\,ADD$	$34\,MUL + 6\,SQR + 25, ADD$
$\mathcal{A} + \mathcal{P} = \mathcal{P}$	$39\,MUL + 3\,SQR + 26\,ADD$	$39\,MUL + 3\,SQR + 23\,ADD$
$\mathcal{A} + \mathcal{R} = \mathcal{R}$	$36\,MUL + 4\,SQR + 31\,ADD$	$25\,MUL + 8\,SQR + 18\,ADD$
$\mathcal{A} + \mathcal{A} = \mathcal{A}$	$1\,INV + 21\,MUL + 3\,SQR + 30\,ADD$	$1\,INV + 17\,MUL + 5\,SQR + 21\,ADD$

2.3.5 Divisor Multiplication Complexities

Estimates for divisor multiplication costs are presented in terms of curve operations (divisor additions and divisor doubling) and the corresponding field operations (multiplications MUL, squaring SQR, addition ADD, and inversion INV). For example, divisor addition and doubling in projective coordinates cost $49MUL + 4SQR + 45ADD$ and $38MUL + 7SQR + 36ADD$ operation counts, respectively. The cost of conversion back to affine coordinates is $4MUL+1INV+15SQR$. If mixed coordinates between affine and projective are used, the operation counts for hyperelliptic curve divisor addition and divisor doubling are $39MUL+4SQR+32ADD$ and $38MUL+7SQR+41ADD$, respectively. The classification of the different coordinates of genus 2 curves in characteristic 2 allows a significant number of different complexities in the formula for divisor doubling and addition as listed in Table 2.3 and Table 2.4, respectively. The results for h of degree 2 and degree 1 are summarized in the tables.

Definition 7 ***The*** **hyperelliptic curve discrete logarithm problem (HCDLP)** ***is the following: Let*** C ***be a hyperelliptic curve over a finite field*** $\mathbb{F}$. ***If*** $D_1, D_2 \in \mathbb{J}_C(\mathbb{F})$, ***determine the smallest integer*** k ***such that*** $D_2 = kD_1$, ***if such an*** k ***exists.***

Divisor multiplication for different coordinates in even characteristic with $\mathbb{F}_{2^{83}}$ are given in Table 2.6. The number of cycles required for performing divisor

Algorithm 2.4 Left-to-right binary method for divisor multiplication

```
Input: $D_{\mathcal{A}}$, $D_{\mathcal{N},\mathcal{P},\,\text{or}\mathcal{R}} \in \mathbb{J}_C$, $k = \sum_{i=0}^{l} k_i 2^i$
Output: $D_{\mathcal{N},\mathcal{P},\,\text{or}\mathcal{R}} = kD_{\mathcal{A}}$
1: $D_{\mathcal{N},\mathcal{P},\text{or}\mathcal{R}} \leftarrow \infty$
2: for $i = l - 1$ down to $0$ do
3:     $D_{\mathcal{N},\mathcal{P},\text{or}\mathcal{R}} \leftarrow 2D_{\mathcal{N},\mathcal{P},\text{or}\mathcal{R}}$
4:     if $k_i = 1$ then
5:         $D_{\mathcal{N},\mathcal{P},\text{or}\mathcal{R}} \leftarrow D_{\mathcal{N},\mathcal{P},\text{or}\mathcal{R}} + D_{\mathcal{A}}$
6:     end if
7: end for
8: return $D_{\mathcal{N},\mathcal{P},\text{or}\mathcal{R}} = kD$
```

multiplication is illustrated in Table 2.5. The corresponding number of cycles for performing a divisor addition and a divisor doubling can be used to estimate the cycles required for divisor multiplication. In conclusion, if the genus, characteristic, and degree of $h(x)$ are restricted to special cases and the correct coordinates are chosen, the best performance is produced for divisor multiplication with HECC.

A central ingredient in cryptosystems based on the HCDLP problem in an abelian group is an efficient process for computing kD for $D \in \mathbb{J}_C$ and for large integers k.

$$\underbrace{D + D + \cdots + D}_{k\ times} = kD$$

This operation is called *divisor multiplication,* and dominates the execution time of hyperelliptic cryptosystems. Algorithm 2.4 is given the left-to-right binary method for divisor multiplication. The following chapters discuss algorithms for performing divisor multiplication efficiently in hardware. In conclusion, if the genus, characteristic, and degree of $h(x)$ are restricted to special cases and the correct coordinates are chosen, the best performance is produced for divisor multiplication with HECC.

2.3.6 Hyperelliptic Curve Algorithms

This section provides a brief description of the algorithms used for adding and doubling divisors on $\mathbb{J}_C(\mathbb{F})$. These group operations are performed in two steps. The first is to find a semi-reduced divisor $D' = \text{div}(u', v')$, such that $D' \sim D_1 + D_2 = \text{div}(u_1, v_1) + \text{div}(u_2, v_2)$ in the group $\mathbb{J}_C(\mathbb{F})$. In the second step, the semi-reduced

Table 2.5: Divisor multiplication (kD) time in cycles

Coordinates	Average Cycles ($h_2 \neq 0$)				Average Cycles ($h_2 = 0$)			
	D_{DBL}	D_{ADD}	D_{mADD}	kD if $(k)_2 = 165\,bit$	D_{DBL}	D_{ADD}	D_{mADD}	kD if $(k)_2 = 165\,bit$
Affine $\mathcal{A}$	6853	9030	8233	1,852,984	5387	8427	8465	1,564,881
Projective $\mathcal{P}$	4169	9285	7465	1,436,274	3876	7994	7023	1,283,299
New Weighted $\mathcal{N}$	5243	8345	6239	1,534,726	4035	8003	4365	1,309,949
Recent $\mathcal{R}$	3318	7354	3146	1,140,185	2164	6986	3270	922,091

Table 2.6: Divisor multiplication for different coordinates, in even characteristic, with size of $(k)_2 = 165$ bits

Operation	Coordinates	Average Costs ($h_2 \neq 0$)	Average Costs ($h_2 = 0$)
$D_{\mathcal{P}} = kD_{\mathcal{A}}$	$2\mathcal{A} = \mathcal{P}$, $\mathcal{A} + \mathcal{P} = \mathcal{P}$	$7040\,MUL + 1068\,SQR + 5037\,ADD$	$6567\,MUL + 1221\,SQR + 7556 ADD$
	$2\mathcal{A} = \mathcal{P}$, $\mathcal{P} + \mathcal{P} = \mathcal{P}$	$7628\,MUL + 1152\,SQR + 6015\,ADD$	$7155\,MUL + 1305\,SQR + 7155 ADD$
$D_{\mathcal{N}} = kD_{\mathcal{A}}$	$2\mathcal{A} = \mathcal{N}$, $\mathcal{A} + \mathcal{N} = \mathcal{N}$	$6040\,MUL + 1152\,SQR + 6255\,ADD$	$5815\,MUL + 1473\,SQR + 6199 ADD$
	$2\mathcal{A} = \mathcal{N}$, $\mathcal{N} + \mathcal{N} = \mathcal{N}$	$6964\,MUL + 1152\,SQR + 3462\,ADD$	$6655\,MUL + 1473\,SQR + 5799 ADD$
$D_{\mathcal{R}} = kD_{\mathcal{A}}$	$2\mathcal{A} = \mathcal{R}$, $\mathcal{A} + \mathcal{R} = \mathcal{R}$	$7184\,MUL + 2034\,SQR + 6491\,ADD$	$5723\,MUL + 1807\,SQR + 6191\,ADD$
	$2\mathcal{A} = \mathcal{R}$, $\mathcal{R} + \mathcal{R} = \mathcal{R}$	$6344\,MUL + 1986\,SQR + 7751\,ADD$	$7823\,MUL + 1807\,SQR + 7115\,ADD$
$D_{\mathcal{N}} = kD_{\mathcal{N}}$	$2\mathcal{N} = \mathcal{N}$, $\mathcal{N} + \mathcal{N} = \mathcal{N}$	$9454\,MUL + 1402\,SQR + 8751\,ADD$	$8315\,MUL + 1473\,SQR + 7599 ADD$
	$2\mathcal{N} = \mathcal{P}$, $\mathcal{N} + \mathcal{P} = \mathcal{N}$	$9702\,MUL + 1318\,SQR + 7261\,ADD$	$8731\,MUL + 1305\,SQR + 6199 ADD$
	$2\mathcal{N} = \mathcal{R}$, $\mathcal{R} + \mathcal{N} = \mathcal{N}$	$8866\,MUL + 1654\,SQR + 8173\,ADD$	$7975\,MUL + 1639\,SQR + 6623\,ADD$
$D_{\mathcal{P}} = kD_{\mathcal{P}}$	$2\mathcal{P} = \mathcal{P}$, $\mathcal{P} + \mathcal{P} = \mathcal{P}$	$9620\,MUL + 1318\,SQR + 9753\,ADD$	$9147\,MUL + 1305\,SQR + 7597 ADD$
	$2\mathcal{P} = \mathcal{N}$, $\mathcal{P} + \mathcal{N} = \mathcal{P}$	$9622\,MUL + 1318\,SQR + 7593\,ADD$	$8981\,MUL + 1473\,SQR + 5737 ADD$
	$2\mathcal{P} = \mathcal{R}$, $\mathcal{R} + \mathcal{P} = \mathcal{P}$	$9114\,MUL + 1570\,SQR + 8831\,ADD$	$7307\,MUL + 1223\,SQR + 5703\,ADD$
$D_{\mathcal{R}} = kD_{\mathcal{R}}$	$2\mathcal{R} = \mathcal{P}$, $\mathcal{P} + \mathcal{R} = \mathcal{R}$	$7450\,MUL + 1986\,SQR + 5925\,ADD$	$6477\,MUL + 1555\,SQR + 5205\,ADD$
	$2\mathcal{R} = \mathcal{N}$, $\mathcal{N} + \mathcal{R} = \mathcal{R}$	$9362\,MUL + 1486\,SQR + 7417\,ADD$	$8555\,MUL + 1391\,SQR + 7199\,ADD$
	$2\mathcal{R} = \mathcal{R}$, $\mathcal{R} + \mathcal{R} = \mathcal{R}$	$8712\,MUL + 1986\,SQR + 4763\,ADD$	$7823\,MUL + 1973\,SQR + 3795\,ADD$
$D_{\mathcal{N}} = kD_{\mathcal{P}}$	$2\mathcal{P} = \mathcal{N}$, $\mathcal{N} + \mathcal{N} = \mathcal{N}$	$9370\,MUL + 1318\,SQR + 7257\,ADD$	$8897\,MUL + 1305\,SQR + 5355 ADD$
	$2\mathcal{P} = \mathcal{N}$, $\mathcal{N} + \mathcal{P} = \mathcal{N}$	$9538\,MUL + 1318\,SQR + 7623\,ADD$	$9065\,MUL + 1305\,SQR + 4781 ADD$
$D_{\mathcal{N}} = kD_{\mathcal{R}}$	$2\mathcal{R} = \mathcal{N}$, $\mathcal{N} + \mathcal{N} = \mathcal{N}$	$10289\,MUL + 1318\,SQR + 8253\,ADD$	$9395\,MUL + 1223\,SQR + 7283\,ADD$
	$2\mathcal{R} = \mathcal{N}$, $\mathcal{N} + \mathcal{R} = \mathcal{N}$	$9530\,MUL + 1654\,SQR + 8505\,ADD$	$8639\,MUL + 1223\,SQR + 7619\,ADD$
$D_{\mathcal{P}} = kD_{\mathcal{N}}$	$2\mathcal{N} = \mathcal{P}$, $\mathcal{P} + \mathcal{P} = \mathcal{P}$	$9954 MUL + 1318\,SQR + 7429\,ADD$	$8815\,MUL + 1473\,SQR + 5537\,ADD$
	$2\mathcal{N} = \mathcal{P}$, $\mathcal{N} + \mathcal{P} = \mathcal{P}$	$9870 MUL + 1318\,SQR + 7261\,ADD$	$8899\,MUL + 1305\,SQR + 6939\,ADD$
$D_{\mathcal{P}} = kD_{\mathcal{R}}$	$2\mathcal{R} = \mathcal{P}$, $\mathcal{P} + \mathcal{P} = \mathcal{P}$	$8710\,MUL + 1650\,SQR + 6765\,ADD$	$7569\,MUL + 1555\,SQR + 5373\,ADD$
	$2\mathcal{R} = \mathcal{P}$, $\mathcal{P} + \mathcal{R} = \mathcal{P}$	$7786\,MUL + 1902\,SQR + 6009\,ADD$	$6729\,MUL + 1807\,SQR + 5289\,ADD$
$D_{\mathcal{R}} = kD_{\mathcal{N}}$	$2\mathcal{N} = \mathcal{R}$, $\mathcal{R} + \mathcal{R} = \mathcal{R}$	$9874\,MUL + 1654\,SQR + 7585\,ADD$	$9151\,MUL + 1807\,SQR + 6119\,ADD$
	$2\mathcal{N} = \mathcal{R}$, $\mathcal{N} + \mathcal{R} = \mathcal{R}$	$8698\,MUL + 1486\,SQR + 7417\,ADD$	$7891\,MUL + 1723\,SQR + 6203\,ADD$
$D_{\mathcal{R}} = kD_{\mathcal{P}}$	$2\mathcal{P} = \mathcal{R}$, $\mathcal{R} + \mathcal{R} = \mathcal{R}$	$10206\,MUL + 1654\,SQR + 8747\,ADD$	$8819\,MUL + 1223\,SQR + 5787\,ADD$
	$2\mathcal{P} = \mathcal{R}$, $\mathcal{P} + \mathcal{R} = \mathcal{R}$	$8778\,MUL + 1654\,SQR + 8777\,ADD$	$7307\,MUL + 1223\,SQR + 5703\,ADD$
$D_{\mathcal{A}} = kD_{\mathcal{A}}$	$2\mathcal{A} = \mathcal{A}$, $\mathcal{A} + \mathcal{A} = \mathcal{A}$	$250\,INV + 1068\,SQR + 4366\,MUL$	$250\,INV + 1055\,SQR + 4391\,MUL$

Algorithm 2.5 Cantor's Divisor Addition on the Jacobian of HECC

```
Input: $D_1 = \mathrm{div}(u_1, v_1)$, $D_2 = \mathrm{div}(u_2, v_2)$, $h$, $f \in \mathbb{F}$.
Output: $D_3 = \mathrm{div}(u_3, v_3) = D_1 + D_2$
1: $d = gcd(u_1, u_2, v_1 + v_2 + h) = s_1u_1 + s_2u_2 + s_3(v_1 + v_2 + h)$
2: $u' = u_1u_2d^{-2}$
3: $v' = [s_1u_1v_2 + s_2u_2v_1 + s_3(v_1v_2 + f)]\, d^{-1} \pmod{u'}$
4: $k = 0$
5: while $\deg u'_k > g$ do
6:     $k = k + 1$
7:     $u'_k = \frac{f - v'_{k-1}h - \left(v'_{k-1}\right)^2}{u'_{k-1}}$
8:     $v'_k = \left(-h - v'_{k-1}\right) \bmod u'_k$
9: end while
10: Output $(u_3 = u'_k,\ v_3 = v'_k)$
```

divisor $D' = \mathrm{div}(u', v')$ is reduced to an equivalent divisor $D = (u, v)$ [95].

Group operations on hyperelliptic curves can be performed by using Cantor's algorithm for adding and doubling and Gauss's algorithm for reducing divisors. These steps require the use of polynomial arithmetic for polynomials that have coefficients in the definition field. An alternative approach was proposed by Harley [38]. Harley computed the necessary coefficients from the steps of Cantor's algorithm directly in the definition field without the use of polynomial arithmetic and also made use of computational tricks. The advantage of this approach is a faster execution time. Up to the present, many attempts have been reported in the literature to improve the HECC group operations. The following subsections includes discussion of only the ones most relevant to the contribution presented in this thesis.

2.3.6.1 Cantor's Group Operations on the Jacobian

Until recently, arithmetic transforms in the Jacobian of HECC have been performed using Cantor's method [16], with modifications introduced by Koblitz [48]. Methods of addition and doubling for curves of genus 2 were considered in [61]. Cantor's algorithm is completely general and holds for all genera, all fields, and all valid inputs. Cantor's divisor addition and doubling on the Jacobian of HECC are given in Algorithm 2.5 and Algorithm 2.6, respectively. All of the explicit formulas have been derived from these algorithms. Nevertheless, this algorithm is too slow, mainly because it employs the greatest common

Algorithm 2.6 Cantor's Divisor Doubling on the Jacobian of HECC

```
Input:  $D_1 = \text{div}(u_1, v_1)$, $h$, $f \in \mathbb{F}$.
Output:  $D_3 = \text{div}(u_3, v_3) = 2D_1$
1: $d = gcd(u_1, 2v_1 + h) = s_1u_1 + s_3(2v_1 + h)$
2: $u' = u_1^2 d^{-2}$
3: $v' = [s_1u_1v_1 + s_3(v_1^2 + f)]\, d^{-1} \,(\text{mod} u')$
4: $k = 0$
5: while  deg $u'_k > g$  do
6:     $k = k + 1$
7:     $u'_k = \frac{f - v'_{k-1}h - \left(v'_{k-1}\right)^2}{u'_{k-1}}$
8:     $v'_k = \left(-h - v'_{k-1}\right) \text{mod} u'_k$
9: end while
10: Output $(u_3 = u'_k,\ v_3 = v'_k)$
```

Algorithm 2.7 Harley Divisor Addition on the Jacobian of HECC

```
Input:  $D_1 = \text{div}(u_1, v_1)$, $D_2 = \text{div}(u_2, v_2)$
Output:  $D_3 = \text{div}(u_3, v_3) = D_1 + D_2$
1: $k = \frac{f - v_1h - v_1^2}{u_1}$
2: $s \equiv \frac{v_2 - v_1}{u_1} \text{mod } u_2$
3: $z = su_1$
4: $u' = \frac{k - s(z + h + 2v_1)}{u_2}$
5: $v' = -(h + z + v_1) \text{mod } u'$
6: Output $(u_3 = u',\ v_3 = v')$
```

divisor (GCD) algorithms and uses up too much memory for restricted environments such as smart cards. To obtain efficient formulas for this work, Cantor's algorithm was used but was restricted to a special case of genus 2, in even characteristic. For example, with $D_1 = D_2$ taken in order to produce an efficient doubling formula. The efficiency of the formulas also depends heavily on the degree of the polynomial $h(x)$ in the curve equation.

2.3.6.2 Harley's Group Operations on the Jacobian

The first attempt to avoid using Cantor's algorithm to achieve faster algorithmic time and to derive explicit formulas was made by Harley [38]. The author noticed that one can reduce the number of operations by distinguishing between possible cases according to the properties of the input divisors. Harley's divisor addition and divisor doubling on the Jacobian of HECC are given in Algorithm 2.7 and Algorithm 2.8, respectively. His approach was then generalized to even charac-

Algorithm 2.8 Harley's Divisor Doubling on the Jacobian of HECC

```
Input: $D_1 = \mathrm{div}(u_1, v_1)$
Output: $D_3 = \mathrm{div}(u_3, v_3) = 2D_1$
1: $k = \frac{v_1^2 - v_1 h - f}{u_1}$
2: $s = \frac{k}{h+2v_1} \bmod u_1$
3: $u' = s^2 + \frac{k - s(h+2v_1)}{u_1}$
4: $v' = -(h + su_1 + v_1) \bmod u'$
5: Output $(u_3 = u', v_3 = v')$
```

teristics by Lange [56]. Harley's study triggered other research, which eventually led to improvements and extensions to the algorithm [41, 79, 57, 88, 87, 60]. The work reported in [97, 41, 88] obtained an even greater increase in speed using Montgomery's trick to reduce the number of inversions to 1. Lange [61] generalized the setting in order to deal with an even characteristic as well through the determination of the exact number of operations needed for addition and doubling to be performed in the most common cases. This algorithm drastically reduced the cost of calculating divisor addition and doubling by specifying the genus of the curves. Most recently, an approach based on parallelization for fast implementation has been proposed by Mishra et al. [72]: they parallelized the affine and the inversion-free formulas of genus 2 curves. The Harley algorithm has long calculation procedures and involves a large number of intermediate variables. In spite of this disadvantage it has potential, but its area and power hardware implementation has not received enough attention. The main focus of this work is the efficient hardware implementation of HECC processors for inversion-free coordinates over a binary field after the application of several architecture synthesis and optimization techniques. The design is optimized to reduce memory and register requirements.

Chapter 3

Architecture Synthesis and Optimization

3.1 Introduction

A basic architecture synthesis and optimization problem addressed in this thesis is based on four assumptions that can be explained as follows: (i) a circuit is specified through the sequencing of an explicit formula algorithm, (ii) a set of functional resources that are fully characterized in terms of area and execution delays, (iii) a set of constraints, and (iv) storage that is implemented through the use of registers and multiplexer interconnections. In this thesis, modified algorithms for performing behavioral optimization are presented. A low area is achieved through liveness analysis of variables combined with a register allocation process. Minimum power consumption is achieved via forwarding paths through operation scheduling, and high performance is achieved by means of efficient register spilling through storage binding. This work also incorporates area, power and computation time tradeoffs through an implementation optimization process, which maps the optimized datapath and control unit analysis in the form of highly efficient hardware implementations.

The modified register allocation via variable liveness analysis (RAVLA), operation scheduling via forwarding paths (OSFPs), and storage binding via efficient register spilling (SBERS) algorithms are employed to produce efficient explicit formulas for both elliptic and hyperelliptic curve algorithms and the optimized hardware implementations. The entire architecture implementation has been synthesized and simulated to present results as tradeoffs among area, power, and computation time. The challenge associated with specification optimization is to determine the appropriate number of registers, their sizes, and their connections to the finite field arithmetic unit (FFAU). Heuristics for allocating variables into multiple distinct single-ported register files are presented in [53]. These heuristics are applicable only to a straightline without branches, which is also the case in the explicit formulas for elliptic and hyperelliptic algorithms. However, the models presented in this section are more applicable for elliptic and hyperelliptic algorithms with arbitrary forwarding paths, variable liveness analysis, and conditional spilling between register and memory binding. Furthermore, for all cases of hyperelliptic curve algorithms, it has been demonstrated that these algorithms are capable of minimizing the number of registers in the register file, which in turn, decreases the computation time and reduces power consumption.

3.2 Design Flow of the Behavioral Architecture Synthesis

Synthesis is the automatic mapping from a high-level description to a low-level description (e.g., gates to transistors, VHDL to gates). Synthesis is important because it increases designer productivity and allows exploration of the design space. Depending on the input and the output descriptions, synthesis comes in two main flavors: logic synthesis and architectural synthesis [83]. Architectural synthesis, also called high-level synthesis, has the primary goal of translating a description of circuit behavior into an architecture. It differs from logic synthesis, which is the specification of Boolean equations that describe a number of operations to be mapped into gates, but not the exact clock cycle based on which the architecture synthesis is specified.

With an architecture synthesis method, the design task is divided into the specification optimization (behavioral) and implementation optimization (structural) phases. The specification synthesis optimization process consists of three sub phases: register allocation, operation scheduling and storage binding. Conventionally, specification synthesis optimization has the goal of decreasing computational time for a given set of resources; however, it has become necessary to develop synthesis optimization techniques whose hardware design also accounts for the minimization of the area and the dissipation of power. Implementation optimization is the initial stage of a hardware design flow that interprets an algorithmic representation of a behaviour and creates a hardware specification that executes the behaviour. More formally, it is the process of creating a structural micro-architectural representation, or register transfer level (RTL) description, from a system specification of an application. A structural micro-architectural representation defines the exact interconnections among a set of architectural resources. An architectural resource is a storage element, a functional unit, or interconnect logic. A storage element provides a method of saving the state of the circuit. A register is an example of such a storage element. A functional unit performs an arithmetic or logic operation (e.g., addition, multiplication, squaring). Interconnect logic is used to route data between the memory and the functional units. For example, a multiplexer propagates a particular piece of data according to its input condition. A control unit issues control signals to direct the resources.

3.2.1 Register Allocation via Variable Liveness Analysis

This section provides a short description of register allocation by means of a variable liveness analysis algorithm. Before an explanation of the algorithm is provided, the following definitions are introduced:

Definition 8 *A variable is said to be defined in a statement in an explicit formula algorithm when a value is assigned to it.*

Definition 9 *A variable is said to be used in a statement in an explicit formula when its value is referenced in an arithmetic expression.*

Definition 10 *A variable is said to be live in a statement if it has been defined earlier and will be used later.*

Definition 11 *The live range of a variable is the execution range between first definition and last use of the variable.*

Definition 12 *The def-use chain of a variable is a directed graph that connects the operations that define a variable to the operations that use this variable. Note that a variable can have multiple definitions and uses.*

Definition 13 *The live time of a variable is defined as the time interval from its definition to its last use. Two variables can share the same register if and only if they do not have overlapping life times.*

Definition 14 *The three operand code of an explicit formula is code in which statements have no more than one destination operand and a maximum of two source operands is said to be three operand code.*

Register allocation is the selection of the appropriate number and type of registers to be used in a given application implementation. The register allocation process attempts to allocate the total number of temporary variables in a three-operand code procedure to the hardware registers in the register file. The temporary variables can be of two types. The first have a value that is used in the same operation statement in which that value is assigned. These variables are implemented as wires or buses in the final implementation. The other type of variable has values assigned in one statement and used in another. The time between the assignment of the value and its last usage defines the lifetime of each variable. In the final implementation, these variables must be mapped to storage units such as register files and memories. The implementation of the task of assigning the variables with non-overlapping lifetimes is by means of register allocation, which directly reduces computational time by converting some of the memory access to register access. Register allocation and operation scheduling are tightly intertwined: one greatly affects the other. They are therefore sometimes performed in different orders. Operation scheduling is discussed in Subsection 3.2.2. Before the strategy used for operation scheduling the model of register allocation must first be defined.

3.2.1.1 Modeling Register Allocation

The problem of minimizing the number of registers required for executing a set of arithmetic formulas has already been studied [72], and the problem is known to be NP-complete. According to the further studies of NP-optimization problems, there is an $O(\log^2 m)$ (where m is the number of operations) approximation algorithm for this problem [72]. In our work, the main emphasis is on exposing a variable liveness analysis among different types of variables in a certain explicit formula. In other words, while ECP or HECP are commonly programmed based on the definition of which operations to execute combined with information about the source of the operands and the destinations of the results, variable liveness analysis is programmed based on the *used* ($\mathcal{U}$) and the *defined* ($\mathcal{D}$) of the intermediate variables. The name of the register allocation via variable liveness analysis comes from the way register allocation are executed which will be explained through the rest of this subsection.

Our motivation is to determine the minimum number of registers needed to store an unbounded number of variables in a certain explicit formula of HECC algorithms based on classifying three types of variables: short, long, and very long lived variables. Achieving the primary goal of this work is thus dependent on determining the answer to the following question: Given an explicit formula $\mathcal{E}$ passed through solving the problem of register allocation via variable liveness analysis model, what is the minimum number of registers $\mathcal{R}$ required in order to store the intermediate variables necessary for executing $\mathcal{E}$ sequentially?

Let $\mathcal{I} = \{I_i\}$, $1 \leq i \leq n$ be the input to $\mathcal{E}$. Assuming that $\mathcal{E}$ is a sequence of arithmetic operations, each has a unique statement $\mathcal{S}$, where $\mathcal{S}_i \ : \ \mathcal{T}_i = \mathcal{P}_i O_i \mathcal{R}_i, 0 \leq i \leq j$, where O_i is one of the finite field operations $\{MUL,\ SQR,\ ADD\}$ and $\mathcal{P}_i, \mathcal{R}_i$ are a subset of $\mathcal{T}_i$. In fact, in the literature, an explicit formula occurs in a multiple operation format. The multiple operation can be converted into a three-operand form $\mathcal{T}_i$ through the use of a simple parser algorithm. A sequence $\mathcal{S} = \{\mathcal{S}_0, \mathcal{S}_1, \cdots, \mathcal{S}_j\}$ of $\mathcal{E}$ will be called a valid sequence if $\mathcal{E}$ can be computed through the execution of its three operand forms $\mathcal{T}_i's$ in the order dictated by the sequence $\mathcal{S}$. For example, if $\mathcal{E} = \{\mathcal{S}_0,\ \mathcal{S}_1,\ \mathcal{S}_2,\ \mathcal{S}_3,\ \mathcal{S}_4,\ \mathcal{S}_5\}$, where

$$\begin{aligned} &\mathcal{S}_0 : \mathcal{T}_0 = x\, ADD\, y; && \mathcal{S}_3 : \mathcal{T}_3 = \mathcal{T}_0\, ADD\, z; \\ &\mathcal{S}_1 : \mathcal{T}_1 = y\, SQR\, y; && \mathcal{S}_4 : \mathcal{T}_4 = \mathcal{T}_3\, MUL\, y; \\ &\mathcal{S}_2 : \mathcal{T}_2 = \mathcal{T}_1\, MUL\, x; && \mathcal{S}_5 : \mathcal{T}_5 = \mathcal{T}_2\, ADD\, \mathcal{T}_4. \end{aligned} \tag{3.1}$$

then only four sequences are valid based on data dependency problem:

$$\begin{aligned} &\{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_0, \mathcal{T}_3, \mathcal{T}_4, \mathcal{T}_5\}, \{\mathcal{T}_0, \mathcal{T}_3, \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_4, \mathcal{T}_5\}, \\ &\{\mathcal{T}_0, \mathcal{T}_3, \mathcal{T}_4, \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_5\}, \{\mathcal{T}_1, \mathcal{T}_0, \mathcal{T}_3, \mathcal{T}_2, \mathcal{T}_4, \mathcal{T}_5\}. \end{aligned} \tag{3.2}$$

The explicit formula $\mathcal{E} = \{S_0,\ S_1,\ S_2,\ S_3,\ S_4,\ S_5\}$ may not be executed in any other order. For the purposes of the work presented in this thesis, the only knowledge of interest is the determination of which valid sequence needs the minimum number of registers when the variable liveness analysis are used for defining different types of variables in the explicit formula $\mathcal{E}$. In $\mathcal{E}$, specific computations can be performed as a result of the set of inputs $\mathcal{I}$ to $\mathcal{E}$, e.g., $\mathcal{T}_0$ and $\mathcal{T}_1$ in the previous sequence. Executing one or more of these computations produces intermediate values that can trigger further operations of $\mathcal{E}$, e.g., $\mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4,$ and, $\mathcal{T}_5$ in the previous sequence example. If $\mathcal{V}$ is the set of computations in $\mathcal{E}$, which can be computed directly from the set $\mathcal{I}$ of inputs to $\mathcal{E}$, and $\mid \mathcal{V} \mid = \alpha$ is the size of the set $\mathcal{V}$, the execution of $\mathcal{E}$ can begin starting from any one of these α operations. Among these operations are three kinds of intermediate variables: short, long, and very long lived variables. An example of a short variable is the computation operation $\mathcal{T}_1$ which immediately triggers the computation operation that is to be followed and that is not needed in other subsequent operations. On the other hand, $\mathcal{T}_2$ is a long lived, which must be hold for a several computation time till it can be used by the computation operation $\mathcal{T}_5$.

An example is provided by a computation flow graph as shown in Figure 3.1 of the following type: there is a root state, the number of states at the first level is α, and each of the first level states correspond to one of the operations in $\mathcal{V}$. The sub-graph rooted at the first level state corresponds to the computation flow graph for completing the remainder of the explicit formula after the operation corresponding to this state has been executed. The leaf states of the computation control flow graph correspond to the last operation in the particular sequence of operations. Clearly, all possible valid computation sequences for completing $\mathcal{E}$ are described by all possible paths from the root to the leaf states. A variable $\mathcal{T}_1$ is live at a statement $\mathcal{S}_0$ if there is a path in the flow graph from this statement to a statement $\mathcal{S}_2$ such that $\mathcal{T}_1 \in \mathcal{I}[\mathcal{S}_2]$ and for each $\mathcal{S}_0 \leq \mathcal{S} < \mathcal{S}_2 : \mathcal{T}_1 \notin \mathcal{I}[\mathcal{S}]$. Liveness analysis focuses on analyzing a flow graph in order to determine which variable places are or are not live. For each flow graph state, two sets can be derived: 1) ***Live-in:*** $\mathcal{I}[\mathcal{S}]$ gives all variables $\mathcal{T}$ that are live before the execution of statement $\mathcal{S}$; 2) ***Live-out:*** $\mathcal{U}[\mathcal{S}]$ gives all variables $\mathcal{T}$ that are live after the execution of statement $\mathcal{S}$.

The basic methodology of the work presented in Figure 3.1 is essentially based on a depth search of a valid sequence that requires a minimum number of registers in order to complete $\mathcal{E}$ and also a valid execution sequence that determines a maximum number of short lived variables for the efficient use of the forwarding paths later through the operation scheduling model. To reduce the number of registers the following four strategies were adopted:

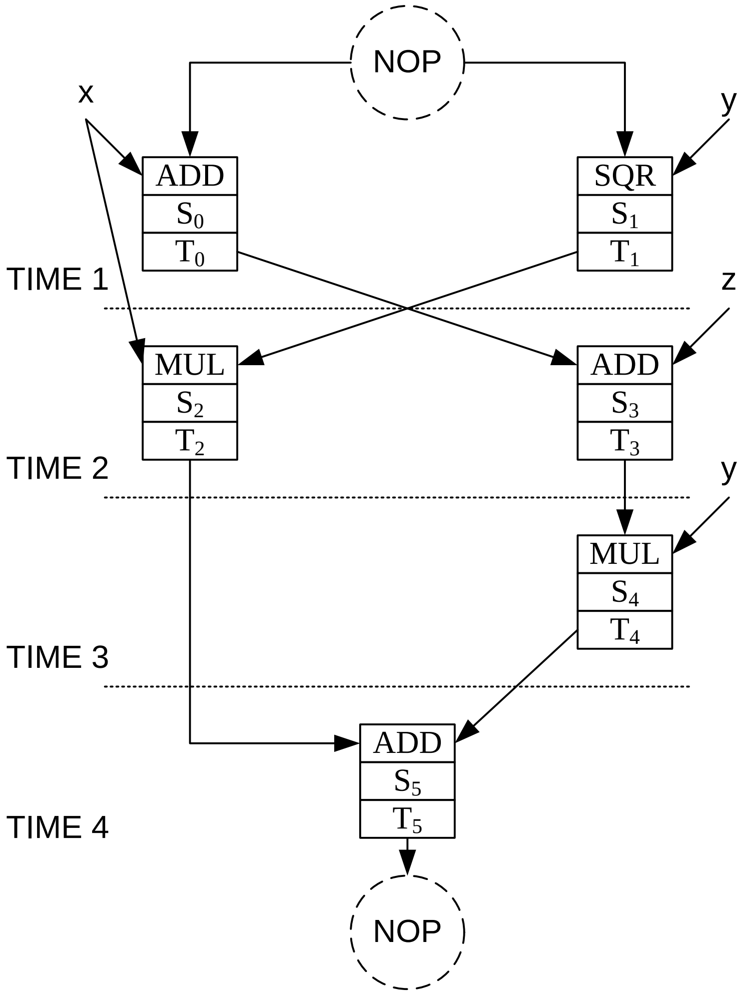


Figure 3.1: Computation flow graph

1. *Neglecting the paths that require the same number of long variables:* As the first valid sequence is obtained, the number of long variables required in order to execute $\mathcal{E}$ according to that sequence is counted and stored it in a register. During the process of looking for another path, the size of the set of long variables is checked after each step. If the current size is equal to the value stored in $\mathcal{R}$, then no further progress along this path is required, and another path is caught. If a valid sequence requires fewer than $\mathcal{R}$ long variables, then the value of $\mathcal{R}$ by is replaced with this new value.

2. *Computing both the input and output states at each statement:* The computation of the number of variables in each input or output sets of each statement of the sequence relies on the following properties: 1) If $\mathcal{T}$ is used in statement $\mathcal{S}$, then $\mathcal{T} \in \mathcal{I}[\mathcal{S}]$. 2) If $\mathcal{T}$ is live after the execution of $\mathcal{S}$ and $\mathcal{T} \notin \mathcal{U}[\mathcal{S}]$, then $\mathcal{T}$ is live before the execution of $\mathcal{S}$. 3) In each statement $\mathcal{S}$, the formal definition of the liveness of variable $\mathcal{T}$ is $\mathcal{T} \in \mathcal{I}[\mathcal{S}]\ O\ \mathcal{T} \in \mathcal{U}[\mathcal{S}] \Rightarrow \mathcal{T} \in \mathcal{U}[\mathcal{S}]$.

3. *Avoiding register writing for short lived variables:* Storing the number of short live variables at each step of the sequence is an area-consuming operation. If the number of long variables stored in $\mathcal{E}$, then each long variable need a register $R \in \mathcal{R}$ in which to be stored. The area can be reduced if the values of short variables are not stored in the sequence, but the register writing is bypassed through the use of forwarding paths. The number of registers needed can therefore not be more than $\mathcal{R}$.

4. *Minimizing the number of registers by coalescing:* If no conflict exists between the liveness of two long variables in the same sequence for two different statements, the same register can be used for both the two long variables, and coalescing can then occur in different situations as long as there is no interference.

The following chapters demonstrate that the implementation of these optimization techniques via a register liveness analysis methodology results in a significant area savings.

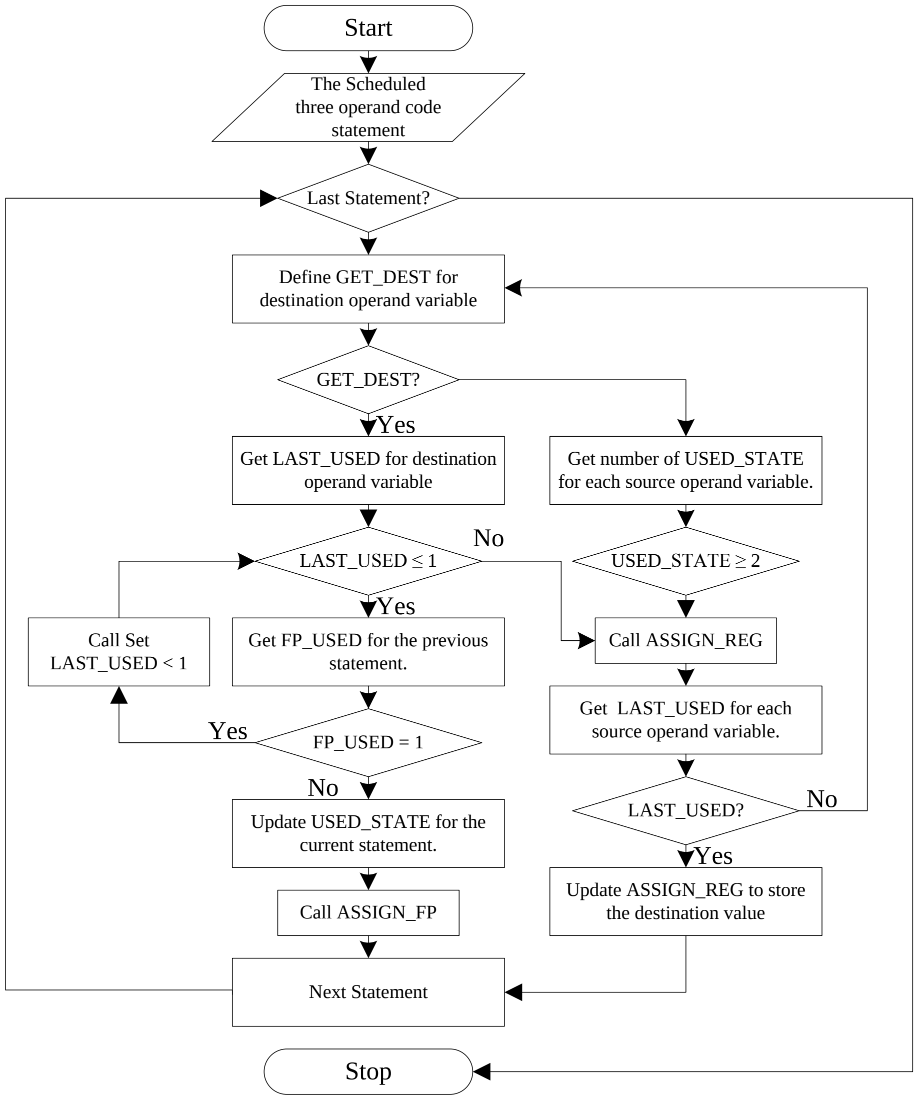

Start
The Scheduled three operand code statement
Last Statement?
Define GET_DEST for destination operand variable
GET_DEST?
Yes
Get LAST_USED for destination operand variable
Get number of USED_STATE for each source operand variable.
No
LAST_USED ≤ 1
USED_STATE ≥ 2
Yes
Call Set LAST_USED < 1
Get FP_USED for the previous statement.
Call ASSIGN_REG
Get LAST_USED for each source operand variable.
Yes
FP_USED = 1
No
LAST_USED?
No
Update USED_STATE for the current statement.
Yes
Update ASSIGN_REG to store the destination value
Call ASSIGN_FP
Next Statement
Stop

Figure 3.2: The procedure for register allocation via variable liveness analysis

3.2.2 Operation Scheduling via Forwarding Paths

Scheduling determines the temporal order of operations. Given a set of explicit formulas with execution delays and a partial ordering, the scheduling model assigns a start time for each operation. The start times must follow the precedence constraints as specified in the system specifications. Additional restrictions, such as timing and area constraints, may be added to the problem, depending on the target architecture. The scheduling affects resource allocation and vice-versa. Therefore, the ordering of these two tasks is sometimes interchangeable; some synthesis tools perform scheduling, then resource allocation, while others allocate the resources first and then schedule the operations. For the work presented in this thesis, the order followed is to perform the register allocation via variable liveness analysis (RAVLA) and then operation scheduling via forwarding paths (OSFPs) .

Schedulers fall into two broad families: unconstrained or constrained operations [69]. Because the number of clock cycles entailed inconsistently finite field arithmetic the focus of this study was on constrained operations in order to minimize the number of registers, without limiting timing. Once the number of live registers is counted and the register allocation is computed the operations are scheduled. This option is the simplest solution and produces good results. A schedule that minimizes both time and area must be chosen. Scheduling assign operations to clock cycles involves two fundamental steps: register constrained and timing constrained. Given an explicit formula for a specific elliptic or hyperelliptic curve algorithm, the clock cycle time for each finite field arithmetic, the resource count, and the resource delays, the register constrained step finds the minimum number of clock cycles needed to execute the explicit formulas. The timing constrained step attempts to determine the minimum number of registers required in order to schedule the operations. However, other multivariate objective functions are also possible, including power, energy, and area.

3.2.2.1 The Proposed Scheduling Model

In this work, the main emphasis is to expose a forwarding path between the functional units (e.g., the finite field squarer, multiplier, and adder) and the register file. In other words, while HECP are commonly programmed based on the definition of which operations to execute combined with information about the source of the operands and the destinations of the results, forwarding path is programmed based on the definition of the transports of the operands and the results. The name of the forwarding path comes from the way operations are executed: the operand data can be read from the output ports of one functional unit transported directly to the source of second functional units. Using a forwarding path enables specific optimizations, such as "dead result read elimination" which can lead to reduced register file pressure.

This subsection presents a formal model for operation scheduling via forwarding paths. Given an explicit formula $\mathcal{E}$, a set of operations $O = \{o_0, o_2, \ldots, o_n\}$, and a collection of computational FFAU units $\{MUL, SQR, ADD\}$ denoted by $r_j = \{r_1, r_2, r_3\}$ respectively, each computational unit $r_j(A_j, C_j, O_{kj})$ has area A_j, a computational time C_j, and can execute only the operations O_{kj}, where $O_{kj} \subset O$ and $\cup_j O_{kj} = O$. Furthermore, each operation in the explicit formula $\mathcal{E}$ must be executable on at least one type of FFAU unit and using at most three registers. A group of registers is defined as $R = \{R_0, R_1, R_2, \ldots R_w\}$. Each register is used for storing an intermediate operation source and the destination variables. Having each operation uniquely associated with one computational unit is called *homogeneous scheduling*. If an operation can be performed by more than one computational unit, the process is called *heterogeneous scheduling* [83]. An FFAU is an example of a computational unit that can perform only one type of operation, so homogeneous scheduling is therefore implied. A further assumption is that the computational cycle delays associated with each operation in different types of computational units have different values, i.e., C_j is a function of the computational unit j. The final assumption is that the execution of the operations is non preemptive; i.e., once the execution of an operation begins, it finishes without interruption.

The solution to the scheduling problem is a vector of statements for each explicit formula denoted as $S = \{S_0, S_1, \ldots, S_k\}$. The end time of each operation is also defined for each statement $E = \{e_0, e_1, , \ldots, e_k)$ reached after a specific delay time $C = \{C_0, C_1, \ldots, C_k\}$. The scheduling problem is formally defined as the minimization of R, which is the number of registers, with respect to the following conditions:

1. An operation can start only when its predecessors have finished.

2. At any given cycle t, the number of computational units working is less than $r_j = \{r_1, r_2, r_3\}$.

3. The register $R = \{R_0, R_1, \ldots R_w\}$ read operation is performed only if the value from the register will be used.

4. The scheduling statements are constructed so that the computational unit reads the operands from the bypasses, rather than from the registers.

The following subsections explain how significant additional reductions in the register file area can be obtained based on scheduling operations that transfer the operands via bypasses rather than reading from the register file. The bypass-aware operation scheduling heuristics that vary in time complexity have been developed, and their effectiveness in reducing register file area has been studied. Operation scheduling is a behavioural view of architecture synthesis, which according to the operations performed in each clock cycle is specified by means of explicit modeling of the units in the datapath synthesis. Since the registers that form the register file are read only when the source operand value is not in the bypasses, such scheduling can reduce both register usage, and consequently register file power consumption. Bypass-aware operation achieved in hardware by using multiplexer. The form of multiplexer used has each data input paired with an enable. Only one enable input is asserted at any time. The data at the data input whose associated enable is asserted appears at the multiplexer's output.

Figure 3.3 shows the procedure for determining exactly when an instruction bypasses the results, which source operands can read them, and when and where the result is written back into the register file. In architecture designs that have complete bypasses, two functions, LAST_USED and FP_USED, are required for each operation: LAST_USED is the number of cycles after issuing the statement, which is the last statement used for the destination operand variable, and FP_USED means that the result is written in the bypasses. In completely bypassed scheduling, the destination of an operation statement is available to every source operand as long as LAST_USED $= 1$, and when cycle $1 <$ LAST_USED $< \frac{j}{2}$, where j is the total number of statements, the destination result is available from the register file until it is overwritten. A cost function, ASSIGN_FP, was used as a means of providing the number of operands of statement S that are read from bypasses. Another cost function incorporated into the new algorithms is WRITE_REG, which simply computes the number of register accesses used through a specific scheduling step. The function Get_OPR is used to define the statement operator either it is MUL for finite field multiplication or it is ADD/SQR for finite filed addition/squarer. The reason for that is whenever it is MUL the algorithm must check FP_USED function in order to avoid perform forwarding paths between two finite field multiplication operations. Because this model was developed for used later in elliptic and hyperelliptic curve algorithms, Get_OPR will only take three values MUL , SQR and ADD for finite field multiplication, squaring, and addition respectively. The function USED_STAT is used to determine the number of statement used by a certain define variable in order to decide weather it will be assign to forwarding paths or not. As long as USED_STAT = 1, which mean a short live variable, the destination will be assigned to forwarding path.

3.2.3 Storage Binding via Efficient Register Spilling

This subsection, describes the concept of storage binding, which is a key process in architecture synthesis. This thesis modified an approach that uses efficient method for mapping variables into register files and memory. The storage binding process binds the variables to the storage units, such as the register file and memory. A recent trend for designers to use register files other than isolated registers in the storage binding. Approaches to storage binding using isolated registers are too complicated and time consuming and are not feasible for large-scale designs. In this work, rather than relying on isolated registers the implementations presented in the following chapters are based on register files and memories as storage units because register files and memories can be more structured, modular, and dense and because their regular layout structure requires less chip area in Application-specific integrated circuit (ASIC) and less resources in Field programmable gate array (FPGA) designs.

The methodology of the architecture synthesis explicitly models the storage binding via efficient register spilling, which is the assignment of each variable to a specific hardware component: explicit mapping between registers and memory. The goal of storage binding via efficient register spilling (SBERS) is to minimize the area by allowing multiple certain type of variables that has been determined through variable liveness analysis to be stored into memory instead of register. The efficient operation scheduling via forwarding paths limits the number of register spilling that is possible.

For low power storage binding that exploits optimal processor structure, the earlier the storage binding model is performed, the higher the dynamic power

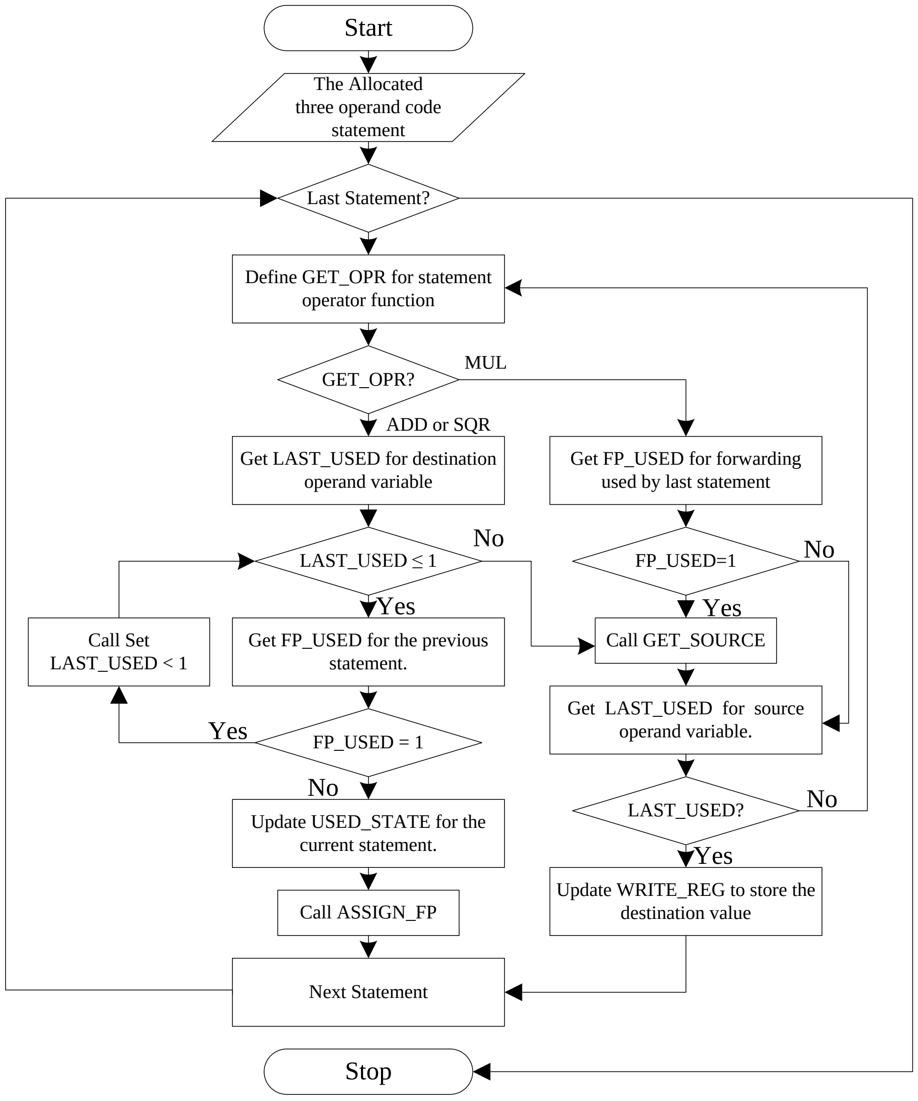

Start
The Allocated three operand code statement
Last Statement?
Define GET_OPR for statement operator function
GET_OPR?
MUL
ADD or SQR
Get LAST_USED for destination operand variable
Get FP_USED for forwarding used by last statement
LAST_USED ≤ 1
No
Yes
FP_USED=1
No
Yes
Call Set LAST_USED < 1
Get FP_USED for the previous statement.
Call GET_SOURCE
Get LAST_USED for source operand variable.
Yes
FP_USED = 1
No
LAST_USED?
No
Yes
Update USED_STATE for the current statement.
Update WRITE_REG to store the destination value
Call ASSIGN_FP
Next Statement
Stop

Figure 3.3: The procedure for bypass-aware operation scheduling via forwarding path

reduction achieved. This section focuses on the reduction of the switching activity in high-level synthesis, especially in the problem of very long lived variable binding. An effective register spilling algorithm has been developed based on the register allocation via variable liveness analysis method so as to reduce switching activity. The new model is based on the determination of very long lived variable type satisfying the conditions LAST_USED$\geq \frac{j}{2}$ and USED_STATE≤ 2 which has been defined through RAVLA of the original explicit formula, and as a result, it produces optimal or close-to-optimal variable binding solutions with faster computing times and a minimum number of registers.

Based on the modeling solutions presented in the previous sections and the functional unit constraints, three novel algorithms have been developed for the modified explicit formula which will be generated in the later chapters for both ECC and HECC operation algorithms: (i) a combined register allocation algorithm with variable liveness analysis, which determines the minimum number of registers need and defines the three types of variables: short, long, and very long lived variables; (ii) a forward datapath scheduling algorithm, which takes advantage of operation slack in order to accommodate the write-port restriction of the register files. (iii) a simultaneous binding algorithm, which determines the assignment of operations into functional units, with a focus on optimizing multiplexers; The scheduling limits possible resource bindings. For example, operations that are scheduled at the same time cannot share the same resource. To be more precise, any two operations can be bound to the same resource only if they are not executed concurrently, i.e., are not scheduled in overlapping time steps. Some resources are capable of executing different operations; e.g., both an addition and a subtraction can be bound to the same arithmetic logic unit. The storage binding can have a significant effect on the area and latency of the circuit because it dictates the number of interconnection logic and storage elements of the circuit. The more hardware that is allocated to the architecture, the more parallel operations can be performed. However, this consumes more areas.

3.3 Optimization Techniques for Architecture Synthesis

3.3.1 Datapath Analysis

The architecture synthesis model can be described as a finite state machine with datapath (FSMD) . FSMD is often referred to as the register-transfer level (RTL), which is composed of a finite state machine that controls the design flow and a datapath that performs operations. The models considered here abstract the information represented by VHDL. Abstract models of behaviour at the architectural level are expressed in terms of operations and their dependencies, which arise for several reasons, the first of which is the availability of data. When an input to an operation is the result of another operation, normal operation depends on the latter. A second reason is the serialization constraints in the specification. A task may have to follow a second one regardless of data dependency. This type of model implicitly assumes the existence of variables whose values store the information that is required and generated by the operations. Each variable has a lifetime, which is the interval from its birth to its death: the former is the time at which the value is generated as an output of an operation, and the latter is the latest time at which the variable is referenced as an input to an operation. The model assumes that the values are preserved in registers during their lifetime.

3.3.2 Control Unit Analysis

This section explains the most difficult part of architectural synthesis, which establishes that a control unit specification has a crucial concurrency property: liveness. Liveness is defined as a condition that must be satisfied at some point during the interaction between the control unit and the computational unit, which is a block diagram that defines a set of shared resources. A control unit specification is expressed in terms of the liveness properties that the control unit must satisfy. Liveness contrasts with a dependency property, which means a condition that must hold continuously throughout the explicit formula execution. Employing liveness in combination with a dependency property approach to synthesis is the basis for a model of a concurrent algorithm in which operations are executed through the accessing and modification of shared resources.

An operation is designated as a three-operand code that define an active computation. An operation computation generally requires the allocation of shared resources.

3.3.2.1 Controllers via Resource Liveness Analysis

The liveness analysis performed in this work is built around the concept of the busy state of the resources encapsulated in the computational unit. The analysis is formulated in terms of a set of variables and a set of operations that can access and modify those variables. Liveness analysis of the resource controllers is used to definite the set of clauses that define the state transitions performed by each operation. The state transition provides the liveness conditions that govern the execution of the explicit formula statements. Through the following chapters we will use both datapath and control unit analysis to improve the design process for hardware implementation of cryptographic algorithms.

3.4 Architecture Optimization Tradeoffs

Architectural optimization comprises architectural synthesis, which includes register allocation, operation scheduling, and resource binding. Complete architectural optimization is applicable for circuits that can be modeled by sequencing (or equivalent) graphs without a start time or binding annotation. The goal of architectural optimization thus consists of the determination of a register allocation γ , an operation scheduling φ, and a resource binding β that optimize the objectives (area, time, and power). The optimization of these multiple objectives can be reduced to the evaluation (or estimation) of the corresponding objective functions. An architectural optimization problem is often solved through the exploration of the area/power tradeoff for different values of the cycles. This approach is based on the fact that the time may be constrained in order to obtain one specific value, or multiple values in an interval, because of system design considerations. The area/power tradeoff can then be optimized through the solving of the appropriate register allocation problems. Other important approaches are the search for the time/power tradeoff for some of the resource binding, and the area/time trade-off for some of the operation schedules. These tradeoffs are important for finding a partial optimization solution when consideration includes scheduling after allocation or binding and vice verse. Unfortunately, the area/power tradeoff for some values of time as well as the time/power trade-off for some values of area are complex problems to solve, because several allocations correspond to a given area and several schedules to a specific time value.

3.4.1 Area/Power Optimization

For a specific execution time, the area depends on register usage. The register allocation problem provides the framework for determining the area/power tradeoff. The minimum power register allocation can be determined through the solving of register liveness analysis problems for different values of the area constraints. The ideal tradeoff curve in the area/power plane monotonically increases in one parameter as a function of the other. The problem becomes more complicated when it must be solved for a circuit with a more complex dependency on operation scheduling. If it is first assumed that the control unit and the wiring have a negligible impact on area and power, then only registers and multiplexers must be taken into account jointly with the computation logic of the operations. Area and power can be determined through register allocation and operation scheduling because the number of registers allocated depends on the dependency of the operations. For a fixed operation scheduling, the number of registers and multiplexers depends on the register allocation. Their area must be added to the computation logic areas when the power is determined. The register liveness analysis problem may thus affect the register allocation, which can result in a smaller area as well as minimum power. These corrections show the circular dependency of operation scheduling, register allocation, and the estimation of area and power.

In practice, CAD tools for architectural optimization perform either operation scheduling followed by register allocation or vice verse. The area and the power are estimated prior to the completion of these tasks. Performing scheduling before allocation permits characterization of the multiplexers and a more precise evaluation of the areas. No dependency constraints are required for the allocation because the operation dependency is determined by the scheduling. In this case, operation scheduling requires that no pair of operations with a dependency execute concurrent. This approach best fits the synthesis of those ASIC circuits that are control dominated and in which the register parameters can be comparable to those of some application specific processors. In the implementation developed in this study, after operation scheduling is performed, the number of clock cycles is fixed before the commencement of the register allocation task. The quality metrics are therefore not the minimum amounts of computational time. The following metrics can be used in order to compare the area/power tradeoffs: the number of clock cycles and the minimum period of usage in one clock cycle.

3.4.2 Power/Time Optimization

In the architectural optimization stage, storage binding has a significant impact on the time constraints between registers. Different storage binding solutions lead to different smallest feasible clock periods, which results in the minimum power consumption. In the new implementation, after the register allocation task, the number of registers in the register file is fixed before the storage binding task is begun. The following metrics can therefore be used to compare power and time: the number of registers in the register file and the size of the multiplexer used in the forwarding paths.

Designs for optimizing power have been an active research area due to the demand for constraint applications. Power reduction techniques have been proposed at all levels of the design hierarchy, from algorithmic and architectural to circuit and technological innovations. It has been shown that for digital hardware design, switching activities represent more than 90 % of the total power consumption [17]. Reducing switching activities is hence a major target in power reduction studies. Pipelining has been used to reduce the critical path computation time and also to reduce switching activities, and hence, power consumption, in digital hardware design. However, the large number of pipelining latches adds a considerable amount of area overhead. Rather than using pipelining, in our new implementation design, power reduction is achieved through developing RAVLA, OSFPS models; through the increasing of the concurrency of the internal finite field operations at the field arithmetic level, and either at the point or the divisor arithmetic level for elliptic and hyperelliptic curves respectively; and through the rearranging of the scalar arithmetic topology from a sequential type to a parallel and forwarding type. Parallel and forwarding type architectures are ideal for low power designs due to their balanced structure, which not only reduces the computation time but also minimizes the switching activities and, hence, the total energy consumption per operation. The two design types shown in Figure 3.4 are examples of architectures for performing three statements over

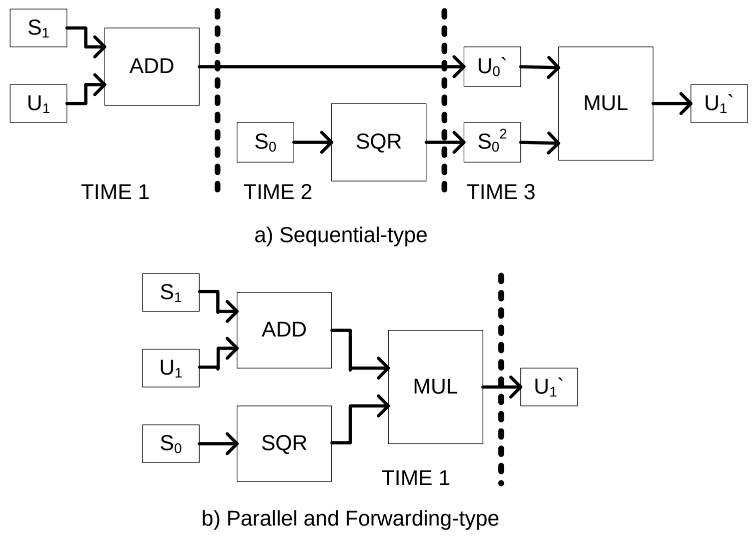


Figure 3.4: Sequential-type versus parallel and forwarding-type architectures

finite field arithmetic as follows:

$$U_0' \leftarrow S_1 + U_1,\ S_0^2 \leftarrow S_0,\ U_1' \leftarrow S_0^2 U_0' \tag{3.3}$$

where $S_1, U_1, U_0', S_0,$ and U_1' are five elements over GF(2^m). Figure 3.4 a) is a sequential type architecture, and Figure 3.4 b) is a parallel and forwarding type. As can be seen, power consumption through each of the arithmetic units is not the same, and each reads from a different register that introduces switching into the circuit. As a result, the power consumption values for subsequent arithmetic units increase. However, in Figure 3.4 b), the critical paths from all input registers to the output register are exactly the same, and the parallel and forwarding type architecture thus consumes approximately 35 % fewer registers than does the sequential type architecture. Parallel and forwarding type architectures can therefore be used to minimize total power consumption without any additional hardware overhead.

Chapter 4

Efficient Implementation for Elliptic Curve Algorithms

Elliptic Curve Cryptography (ECC) was independently proposed in the mid 1980s by Miller [70] and Koblitz [47]. Area and power optimization are two important design issues for Elliptic Curve Cryptography (ECC) used in many embedded systems. One benefit of ECC is that it requires a much shorter key length than other public key cryptosystems to provide an equivalent level of security. However, efficient hardware implementation of an Elliptic Curve Processor (ECP) for lightweight devices is a challenge. In this chapter we propose an efficient processor for ECC that aims to reduce the number of registers compared to those that have appeared in the literature. We take advantage of forwarding paths in the ECP to avoid writing/reading of short-lived variables to/from the register file. The proposed ECP design is implemented over $\mathbb{F}_{2^{163}}$ on Xilinx XC4VLX200 FPGA device to verify its functionality and measure its performance. This work yields an area saving up to 38% in the number of flip-flops and up to 27% with respect to the number of look-up tables (LUTs). The performance overhead is equal to $1.8\,ns$ to be added to the ECP critical path.

4.1 Introduction

Elliptic curve cryptography has occupied the center stage of public key cryptographic research. The main reason behind it is their shorter key for the same level of security due to its ability to provide greater security per bit compared to public key systems such as RSA. ECC has also been included in IEEE P1363

[4] and NIST [75] standards. The traditional way of implementing ECC is software, running on general-purpose processors or on digital-signal processors [51]. Nevertheless, in some applications dedicated hardware must be considered. The designer of elliptic curve hardware implementation is faced with many choices at design time, each of which can impact the performance of the implementation in different ways. There are many examples in the literature of how these design choices can affect the performance of an elliptic curve hardware implementation. The effect of design choices on area, power and energy in elliptic curve hardware has been less well studied. This thesis is concerned with the implementation of an optimized ECP with a special *scalar* multiplier and self-controlled architecture. Its goal is to optimize the ECP implementation targeted for resource restricted environments in terms of hardware usage. In addition the power and energy results are presented and compared.

A number of hardware implementations for standardized elliptic curve cryptography were suggested in the literature [18, 19, 52, 6, 100], but very few of them aimed for low-end devices [30, 40, 65, 99]. Most implementations focus on speed and are only suitable for server end applications due to their huge area requirements [19, 99, 55, 67]. However, there is an equally important need for stand-alone ECP engines in small constrained devices used for different applications, like sensor networks and mobile devices. An application-specific ECP was reported in [18]. The design is based on a combined algorithm to perform point addition and doubling using a multistage pipelined *finite field* multiplier. In [81] a high performance elliptic curve cryptography processor over $\mathbb{F}_{2^{163}}$ was proposed which achieved a high throughput with a twice increase in the hardware complexity. The architecture is based on the López-Dahab elliptic curve point multiplication algorithm with Gaussian normal basis (GNB) for $\mathbb{F}_{2^{163}}$. Moreover, in [55] a high performance elliptic curve processor was presented based on the Montgomery scalar multiplication algorithm. The architecture consists of three Arithmetic Units (AUs). Each AU contains a word-serial multiplier, a squarer and an adder in addition to a bit-parallel modular reduction for the irreducible pentanomials. Pseudo-multi-core ECP over $\mathbb{F}_{2^{163}}$ was reported in [100]. The architecture primarily consists of three finite field (FF) RISC cores with a main controller to achieve pipelining. Each core consists of two 41×163 FF multipliers, two bit-parallel squarer and two adders. The presented architecture is based on Instruction level parallelism (ILP) of paralleled López-Dahab algorithm among

the three FF cores.

The aims of this work are (i) to optimize area requirements by reducing the number of registers in the Register File (RF) of the ECP; (ii) to present a low-power embedded ECP architecture that takes advantage of the "forwarding" ("bypassing") hardware for avoiding the power cost of writing and reading short-lived variables to and from the RF; and (iii) to improve speed, throughput, and critical path in the implementation of ECP which keeping the area requirment low.

This chapter is organized as follows. In section 4.2, we briefly recap previous hardware implementation on ECC and introduce conventional elliptic curve implementation for $\mathcal{LD}$ projective coordinates. Section 4.3 summaries architecture synthesis and optimization techniques used to develop efficient explicit formula for different coordinates systems. Section 4.4 gives synthesis and simulation results and comparisons. Conclusions are provided in section 4.5.

4.2 Previous Work in ECC Hardware Implementation

Concerning the design of lightweight implementations, most research on elliptic curve implementations focused on minimizing the computation time per point multiplication. Several implementations aimed at minimizing power consumption for applications in constrained environments have also been reported [77, 10]. A particularly active area of research is that of efficient hardware implementations of elliptic curve operations, including hardware description language developments, programmable hardware realizations, and fabricated custom circuits.

4.2.1 Low-Power Designs

An ASIC based low power elliptic curve digital signature chip was presented in [85]. Typically, an ASIC based architecture uses one particular algorithm and coordinate system to perform the point scalar multiplication. Bertoni *et al.* [12] presents a low power elliptic curve processor based on an Atmel 8-bit microprocessor and a dedicated ASIC coprocessor to perform field multiplications. Keller *et al.* [43] have studied the effects that algorithm and coordinate choice have on

the power consumption of reconfigurable $\mathbb{F}_{2^m}$ ECP which was implemented using FPGA.

Richard *et al.* [85] introduces a VHDL (stands for VHSIC Hardware Description Language, where VHSIC stands for Very High Scale Integrated Circuit) design that incorporates optimizations intended to provide digital signature generation with as little power as possible. While the value of elliptic curve arithmetic which is enabling public-key cryptography to serve in resource-constrained environments is well accepted, it was motivated by the need to reduce the resources required to provide strong public-key authentication for sensor-based monitoring system and critical infrastructure protection. For these applications, signature generation is often performed in highly constrained, battery-operated environments, whereas signature verification is performed on desktop systems with only the typical constraint of purchasing power. The work of Gaubatz *et al.* [31] discusses the necessity and the feasibility of public key cryptography protocols in sensor networks. In [31], the authors investigated implementations of two algorithms for this purpose: Rabin's scheme and NTruEncrypt. The conclusion is that NTruEncrypt features a suitable low-power and small area with the total of 3000 gates in complexity and power consumption of less than $20 \mu W$. On the other hand, RSA is concluded to be impossible with the imposed constraints. In the follow-up work [32] the authors have compared the previous two with ECC solution for wireless sensor networks. The architecture of the ECP occupied an area of 18,720 gates and consumes less than $400 \mu W$ of power at clock frequency of $500\ kHz$. The field used was a prime field with elements that have bit-lengths of 102. The level security is therefore not so high but it is seems to meet the requirements for lightweight applications.

The work of Goodman and Chandrakasan [34] also dealt with power efficient solutions. They proposed a domain-specific reconfigurable cryptographic processor (DSRCP). The primary component of the DSRCP is the reconfigurable data-path. The data path consists of four major functional blocks: an eight-word register file, a fast adder unit, a comparator unit and the main reconfigurable logic unit. Multiplication is performed using Montgomery multiplication. At $50\ MHz$, the processor operates at supply voltage of 2V and consumes at most $75\ mW$ of power. In ultra-low-power mode the processor consumes at most $525\ \mu W$. The PhD thesis of Goodman [35] offers many useful ideas and considerations for low-power designs.

4.2.2 High performance Designs

Elliptic curve scalar multiplication is a fundamental operation in elliptic curve cryptosystems. Recently, a number of hardware architectures have been proposed in the literature to speed up this operation. Richard *et al.* [85] present a design incorporates built-in optimizations using the point-halving algorithm for elliptic curves and field towers to speed up the finite field arithmetic in general. Further enhancements of basic finite field arithmetic operations intended to provide digital signature generation with low power are also included. Kim *et al.* [44] introduce a hardware architecture to take advantage of a non-conventional basis representation of finite field elements to make point multiplication more efficient. Moon *et al.* [73] address field multiplication and division, proposing a new method for fast elliptic curve arithmetic appropriate for hardware.

Goodman and Chandrakasan [34] tackle the problem of providing energy-efficient public-key cryptography in hardware while supporting multiple algorithms, including elliptic curve-based algorithms. Moving closer to applications of ECC, Aydos *et al.* [9] have implemented an ECC-based wireless authentication protocol that utilizes the elliptic curve digital signature algorithm (ECDSA). Very recently, McLoone and Robshaw [68] reconsider the hardware cost of public-key cryptography for ultra constrained area and power applications. Their implementation allows for a tag that can participate in an authentication protocol a limited number of times. As a result, the tag can store in memory the responses to a limited number of authentications and thus, the hardware performance is increased. Mishra *et al.* [71] propose a pipelining scheme for implementing ECC. They implement two stage pipeline as at any point of time there will be at most two operations in a "Producer-Consumer Relation" or in one step of the three operand code. The one which enters the pipeline earlier will be producing outputs which will be consumed by the second operation as inputs. As soon as the producer process exits the pipeline the subsequent EC-operation will enter the pipeline as the consumer process. The earlier consumer would be producing outputs now which will be consumed by the newer process.

The estimated performance of the ECC processor for the field $\mathbb{F}_{2^{163}}$ is given in [89]. For the irreducible polynomial $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$ one point multiplication takes $163 * 15M = 2445M$. Conversion of coordinates $A \to P$ and $P \to A$ takes respectively $2M$ and $I + 2M$. Assuming that inversion is done by means of Fermat the total for conversion is around $300M$. This all together results

in approximately $3000M$. One filed multiplication (M) takes 163 cycles, which results in 489000 cycles for point multiplication. With a clock frequency of even $1MHz$ one point multiplication would take less than half a second.

Bijan and Hasan [6] proposed a high performance architecture of elliptic curve scalar multiplication using the Montgomery ladder method over finite field $\mathbb{F}_{2^m}$. The idea depends on the parallel execution of multiplication and addition/squaring in one iteration and/or in the entire multiplication loop. They also proposed a pseudo pipelined finite field multiplier with short critical path. The introduced finite field multiplier uses pipelining to reduce the critical path. Kumar and Paar [51] proposed an affine coordinate ASIC implementation of the ECC processor, using $\mathcal{LD}$ projective coordinates a modified Montgomery point multiplication method for binary elliptic curves. An area consumption of 10k to 18k GE is obtained $0.35\,\mu m$ CMOS process.

4.2.3 Compact Implementations

Previous implementations of ECC processors have been based on VLSI chips which implement a coprocessor for performing the underlying field operations. Motorola DSP 56000 was used to implement a complete ECC system which calculate 5 points a second on a super singular curve [5]. In 1993, the same team developed a processor for operations in the $\mathbb{F}_{2^{155}}$ [5] which used 11,000 transistors and could operate at 40 MHz. This implementation was intended to be compact yet secure. A field programmable gate array (FPGA) based processor for elliptic curve cryptography in a composite $\mathbb{F}_{(2^n)^m}$ was developed by Rosner [82]. A compact super-serial multiplier for FPGAs which trades off performance for area was reported in 1999 and its performance for field (polynomial basis) and curve multiplications over $\mathbb{F}_{2^{167}}$ was also presented [76]. ECPs based on serial finite field multiplier on $\mathbb{F}_{2^m}$ are compact processors but slower than other implementations [99, 66].

Low-power and compact implementations become an important research area with the constant increase in the number of hand-held devices such as mobile phones, smart cards, PDAs *etc.* Schroeppel *et al* [85] present a design for ECC over binary fields that was optimized for power, space and time in order to provide digital signatures. Wolkerstorfer [91] shows that ECC based public key cryptography is feasible on RFID-tags by implementing the ECDSA (Elliptic curve Digital Signature Algorithm) on a small chip. Batina *et al* [10] show that

by trading off performance for area it is possible to implement EC public-key cryptography on a tag.

Wolkerstorfer [91] has presented an ECP that able to calculate all arithmetic operations used in ECC: addition, multiplication, and inversion in the finite fields $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$. Bit-serial multiplication is calculated by an improved version of Montgomery's algorithm. A realization of the ECP on $0.35\ \mu m$ CMOS needs $1.31 mm^2$ and can be clocked with $68.5\ MHz$. On a $180\ nm$ CMOS process, the area will shrink to $0.35\ mm^2$which is acceptable for RFID tags.

Yong Ki Lee *et al* [65] propose an architecture for compact elliptic curve multiplication processors using López-Dahab's Montgomery scalar multiplication algorithm [54], and using the projective coordinates system to avoid inverse operations. They show that if López-Dahab's algorithm is implemented based on the modular arithmetic logic unit in a conventional way, the total number of registers is nine. As the registers occupy more than 80% of the gate area in the conventional architecture, reducing the number of the registers and the complexity of the register file are very effective to minimize the total gate area. In order to minimize the system size, they propose a new formula for the common projective coordinates system where all the Z-coordinate values are equal. This resulted in reducing the number of registers by one. Then reducing two more registers by using register reuse technique and designing the compact register file architecture by limiting the access to the registers [64]. Accordingly, three registers were reduced in total to become 6 registers. They also proposed a reduction of the complexity of the register file by designing a circular shift register file. However, the use of this register file increases the number of cycles due to the control overhead. The authors also calculated the number of execution cycles needed to perform point multiplication operation which is equal to 313,901 clock cycles. In this work, we are proposing an optimized architecture for ECP to perform point multiplication operation using 4 registers only without using a circular shift register file to avoid the total overhead. Furthermore, the number of execution cycles needed to perform point multiplication operation is 309,546 clock cycles which is less than that of Yong Ki Lee [65] work. Moreover, the power consumed by the register file is expected to decrease because of the reduction in the number of the register file and the use of bypassing hardware which will reduce the number of write backs to the register file.

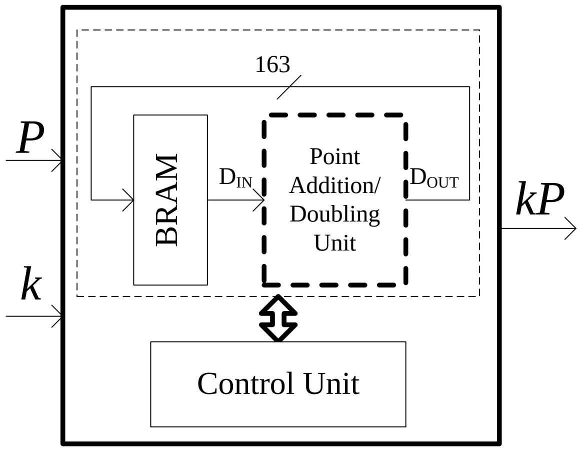


Figure 4.1: A top level architecture for the EC processor

4.3 Architecture Synthesis of ECC processor

This section presents the results obtained from the implementation of an optimized ECP architecture on an FPGA. An ECP is an elliptic curve cryptographic processor using López-Dahab projective coordinate. FPGA implementation that includes consideration of explicit formulas based on López-Dahab projective coordinates. Results of register management for both conventional and optimized point addition/doubling unit for performing point addition/doubling algorithm were proposed. A method for calculating the power dissipated by the register file, bypassing hardware and the arithmetic logic unit in the conventional and optimized schemes are presented. The number of clock cycles needed to perform point multiplication operation in both conventional and optimized approaches were also tabulated. Moreover, a comparison between the optimized approach and the conventional approaches in terms of execution time per clock cycles and power dissipation are presented.

4.3.1 ECC processor on FPGA

The EC processor is, in fact, the hardware architecture for scalar multiplication. The top-level architecture consists of a control unit, block RAM (BRAM), and a point addition/doubling unit, as shown in Figure 4.1. The control unit receives the EC parameters, reads a key (or a scalar), and controls the point addition/doubling unit according to the binary double and add point multiplication algorithm shown in section 2.2. The point addition/doubling unit, on

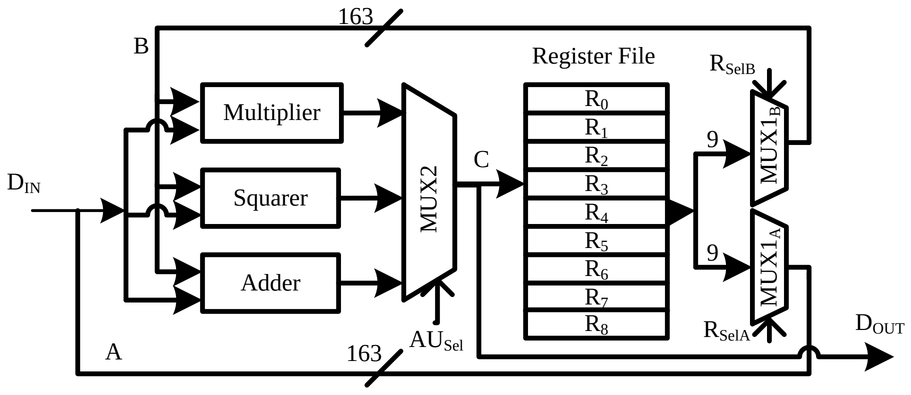


Figure 4.2: conventional point addition/doubling unit

the other hand, is responsible for computing all required field arithmetic operations. At the beginning of the scalar multiplication operation, the BRAM is assumed to contain the scalar and a projective EC point. These values must be maintained during the iterations of the EC scalar multiplication. The point addition/doubling unit designed for computing the scalar multiplication algorithms is discussed in the following sections.

4.3.2 Macroscopic Structural view of Conventional ECC datapath

An example of conventional architecture for point addition/doubling [52, 28, 54] is depicted in Figure 4.2. Its three main units are as follows: an adder, a multiplier, and a squarer, all of which are for $\mathbb{F}_{2^m}$. In the work presented in this thesis, the type of the adder and the squarer are bit parallel, while the type of multiplier is bit serial. These three field arithmetic units are closely interconnected inside a single finite field arithmetic unit (FFAU) and share a common input data bus *A*. A second operand is provided to the FFAU through an additional data bus *B*. The operands are stored in a number of registers, and the output of a register is placed on buses *A* and *B* using a multiplexer (MUX1) with control signals (R_{SelA} and R_{SelB}). The three FAUs are connected to bus *C* via a multiplexer (MUX2) controlled by AU_{sel}. The question is, given an explicit formula for both point addition and point doubling operations, what is the minimum number of registers required to compute the formula sequentially or in parallel.

4.3.2.1 Register-Transfer-Level Design for López-Dahab Algorithm Implementation

Register-transfer-level (RTL) design is a complicated-sounding name for a simple concept. In RTL design a circuit is described as a set of registers and a set of transfer functions that describe the flow of data between the registers. The registers are implemented directly as flip-flops while the transfer functions are implemented as blocks of combination logic.

This stage in the RTL design cycle is commonly referred to as register allocation and operation scheduling described in chapter 3. Register allocation refers to the mapping of data operations onto hardware resources. Operation scheduling refers to the choice of clock cycle during which an operation will be performed in a multi-cycle operation. Registers must also be allocated for all values that cross over from one clock cycle to a later one. Register allocation and operation scheduling are interlinked and must normally be carried out simultaneously. The aim is to maximize resource usage and simultaneously to minimize the number of registers required to storing intermediate results. Owing to its simplicity, the allocation stage can be considered trivial because all multiplications, squaring, and additions must be allocated to the one multiplier, squarer, and adder, respectively. The scheduling operation entails choosing in which clock cycle each multiplication, squaring, and addition is to be performed.

Algorithm 4.1 describes the register-transfer-level design for the López-Dahab mixed coordinates point multiplication operation. The three-operand code, like the register-transfer statements, is shown at each step of the algorithm. Figure 4.2 shows that the algorithm is implemented using nine registers. In fact, this section presents an RTL design for López-Dahab coordinates that are both projective and mixed. Figure 4.2 shows a general design for both mixed and projective coordinates, with the only difference being the number of intermediate values used by the algorithm. For example, with mixed coordinates the intermediate values occupy only registers $R_0, R_1, R_2,$ and $R_3,$ as indicated in Algorithm 4.1. At the beginning of the mixed coordinates point multiplication operation, it is assumed that five read accesses from BRAM are performed in order to store X_1, Y_1, Z_1, X_2, and Y_2 in R_4, R_5, R_6, R_7, and R_8 respectively. If $k_i = 1$, point addition and point doubling are executed. At the end of the point multiplication operation, three write accesses are executed in order to store the resulting point $Q = (X_3, Y_3, Z_3)$. At each step of the register file management, each field operation

Algorithm 4.1 Register management for the left-to-right binary point multiplication in $\mathbb{F}_{2^m}$ using López-Dahab mixed projective coordinates [37].

Input: P, k where $P \in E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + ax^2 + b\,,\ b = 1,\ k = [k_{t-1} \cdots k_1, k_0]_2$.
Output: $Q = k\,P$ where $Q = (X_3, Y_3, Z_3)$ in LD coordinates $\in E(\mathbb{F}_{2^m})$

```
1:  for $i = t - 1$ down to 0 do
2:      $R_0 \leftarrow MUL\,(X_1,\ Z_1)$
3:      $R_1 \leftarrow SQR\,(X_1)$
4:      $R_2 \leftarrow ADD\,(R_1,\ Y_1)$
5:      $R_3 \leftarrow MUL\,(R_0,\ R_2)$
6:      $Z_3 \leftarrow SQR\,(R_0)$
7:      $R_0 \leftarrow SQR\,(R_2)$
8:      $R_0 \leftarrow ADD\,(R_0,\ R_3)$
9:      $X_3 \leftarrow ADD\,(R_0, Z_3)$
10:     $R_0 \leftarrow ADD\,(Z_3,\ R_3)$
11:     $R_1 \leftarrow SQR\,(R_1)$
12:     $R_1 \leftarrow MUL\,(R_1,\ Z_3)$
13:     $R_0 \leftarrow MUL\,(R_0,\ X_3)$
14:     $Y_3 \leftarrow ADD\,(R_0,\ R_1)$
15:     if $k_i = 1$ then
16:         $R_0 \leftarrow SQR\,(Z_1)$
17:         $R_0 \leftarrow MUL\,(Y_2,\ R_0)$
18:         $R_0 \leftarrow ADD\,(Y1,\ R_0)$
19:         $R_1 \leftarrow MUL\,(X_2,\ Z_1)$
20:         $R_1 \leftarrow ADD\,(X_1,\ R_1)$
21:         $R_2 \leftarrow MUL\,(R_1,\ Z_1)$
22:         $Z_3 \leftarrow SQR\,(R_2)$
23:         $R_1 \leftarrow SQR\,(R_1)$
24:         $R_1 \leftarrow ADD\,(R_0,\ R_1)$
25:         $R_1 \leftarrow ADD\,(R_1,\ R_2)$
26:         $R_1 \leftarrow MUL\,(R_2,\ R_1)$
27:         $R_3 \leftarrow SQR\,(R_0)$
28:         $X_3 \leftarrow ADD\,(R_3,\ R_1)$
29:         $R_1 \leftarrow MUL\,(R_0,\ R_2)$
30:         $R_0 \leftarrow MUL\,(X_2,\ Z_3)$
31:         $R_2 \leftarrow ADD\,(R_0,\ X_3)$
32:         $R_1 \leftarrow ADD\,(R_1,\ Z_3)$
33:         $R_2 \leftarrow MUL\,(R_2,\ R_1)$
34:         $R_0 \leftarrow ADD\,(Y_2,\ X_2)$
35:         $R_1 \leftarrow SQR\,(Z_3)$
36:         $R_3 \leftarrow MUL\,(R_0,\ R_1)$
37:         $Y_3 \leftarrow ADD\,(R_3,\ R_2)$
38:     end if
39: end for
40: Return $(Q = (X_3, Y_3, Z_3))$
```

is be performed according to the explicit formulas stated in section 2.3.4.

For example, the operation in step 17 is a field multiplication between two operands Y_2 and Z_1^2 stored at R_0. These two operands are available for the multiplication input with the R_{SelA} and R_{SelB} select control signals. After 163 clock cycles, the result is written back to R_0 as governed by the AU_{Sel} selection control signal, as shown in Figure 4.2. Each field addition and field squaring step is executed in only one cycle. On the other hand, a field multiplication operation requires that 163 cycles be executed, corresponding to the irreducible pentanomial function $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ for the NIST recommended curves $m = 163$ [37].

Similarly, Algorithm 4.2 shows the conventional register management for the point multiplication with López-Dahab projective coordinates. As presented

Algorithm 4.2 Register management for the left-to-right binary point multiplication in $\mathbb{F}_{2^m}$ in López-Dahab projective coordinates [37].

Input: P, k where $P \in E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + ax^2 + b\,,\ b = 1,\ k = [k_{t-1} \cdots k_1, k_0]_2$.
Output: $Q = k\,P$ where $Q = (X_3, Y_3, Z_3)$ in $\mathcal{LD}$ coordinates $\in E(\mathbb{F}_{2^m})$

```
1:  for $i = t-1$ downto 0 do
2:      $R_0 \leftarrow MUL\,(X_{1(3)},\ Z_{1(3)})$
3:      $R_1 \leftarrow SQR\,(X_{1(3)})$
4:      $R_2 \leftarrow ADD\,(R_1,\ Y_{1(3)})$
5:      $R_3 \leftarrow MUL\,(R_0,\ R_2)$
6:      $Z_3 \leftarrow SQR\,(R_0)$
7:      $R_0 \leftarrow SQR\,(R_2)$
8:      $R_0 \leftarrow ADD\,(R_0,\ R_3)$
9:      $X_3 \leftarrow ADD\,(R_0, Z_3)$
10:     $R_0 \leftarrow ADD\,(Z_3,\ R_3)$
11:     $R_1 \leftarrow SQR\,(R_1)$
12:     $R_1 \leftarrow MUL\,(R_1,\ Z_3)$
13:     $R_0 \leftarrow MUL\,(R_0,\ X_3)$
14:     $Y_3 \leftarrow ADD\,(R_0,\ R_1)$
15:     if $k_i = 1$ then
16:         $R_0 \leftarrow MUL\,(X_1,\ Z_2)$
17:         $R_1 \leftarrow MUL\,(X_2,\ Z_1)$
18:         $R_2 \leftarrow SQR\,(R_0)$
19:         $R_3 \leftarrow SQR\,(R_1)$
20:         $R_4 \leftarrow ADD\,(R_2,\ R_3)$
21:         $R_5 \leftarrow SQR\,(Z_2)$
22:         $R_5 \leftarrow MUL\,(Y_1,\ R_5)$
23:         $R_6 \leftarrow SQR\,(Z_1)$
24:         $R_6 \leftarrow MUL\,(Y_2,\ R_6)$
25:         $R_7 \leftarrow ADD\,(R_5,\ R_6)$
26:         $R_8 \leftarrow ADD\,(R_0,\ R_1)$
27:         $R_7 \leftarrow MUL\,(R_7,\ R_8)$
28:         $R_8 \leftarrow MUL\,(Z_1,\ Z_2)$
29:         $Z_3 \leftarrow MUL\,(R_4,\ R_8)$
30:         $R_3 \leftarrow ADD\,(R_6,\ R_3)$
31:         $R_6 \leftarrow ADD\,(R_2,\ R_5)$
32:         $R_2 \leftarrow MUL\,(R_1,\ R_6)$
33:         $R_1 \leftarrow MUL\,(R_0,\ R_3)$
34:         $X_3 \leftarrow ADD\,(R_1,\ R_2)$
35:         $R_3 \leftarrow MUL\,(R_4,\ R_5)$
36:         $R_5 \leftarrow MUL\,(R_0,\ R_7)$
37:         $R_0 \leftarrow ADD\,(R_5,\ R_3)$
38:         $R_2 \leftarrow MUL\,(R_0,\ R_4)$
39:         $R_0 \leftarrow ADD\,(R_7,\ Z_3)$
40:         $R_5 \leftarrow MUL\,(R_0,\ X_3)$
41:         $Y_3 \leftarrow ADD\,(R_2,\ R_5)$
42:     end if
43: end for

44: Return $(Q = (X_3, Y_3, Z_3))$
```

in the algorithm, the intermediate values require that nine registers R_0 to R_8 be stored. The design therefore causes the elements X_1, Y_1, Z_1, X_2, Y_2 to be stored in the BRAM, which, if $k_i = 1$, needs a total of 20 read accesses through D_{IN} and six write accesses to D_{OUT} in one iteration. For example, in step 16 of Algorithm 4.2, a finite field multiplication(MUL) performed between two intermediate results stored in R_4 and R_8 results in an element Z_3, which consumes a write access to BRAM. This element is repeatedly used by the remainder of the point addition steps or by the point doubling steps, as specified in step 30. The notation $Z_{1(3)}$ indicates reuse of the memory location of Z_1 by Z_3 after the first iteration of the point multiplication algorithm is executed which has the value $k_i = 0$. The following section presents the use of the optimization analysis described in chapter 3 in order to reduce the number of registers used for storing the intermediate variables and to reduce the number of read/write accesses that are required from/to BRAM.

4.3.3 Macroscopic Structural view of Optimized ECC datapath

An optimized point addition/doubling unit is illustrated in Figure 4.3. Similar to the conventional point addition/doubling unit described in section 4.3.2, it consists of the three main components: the adder, the multiplier, and the squarer, all of which are for $\mathbb{F}_{2^{163}}$. The enhancement in the new unit is the addition two multiplexers: MUX3 and MUX4 which are used to provide the forward paths from one finite arithmetic unit to another. These multiplexers are controlled by two signals: D_{Sel} and E_{Sel}. The operands are stored in the register file, which consists of four registers whose output is selected for buses A and B using multiplexer MUX1 with control signals R_{SelA} and R_{SelB}. These signals are addressed from the control unit, as shown in Figure 4.1. The reason for using only four registers is the result of the application of the new methodology for register allocation and operation scheduling as explained in chapter 3. The following subsection provides an analysis of the modified left-to-right binary point multiplication in both mixed and projective López-Dahab coordinate systems. This analysis is based on the application of the three models described in section 3.2. The design shown in Figure 4.3 is a general one for both mixed and projective López-Dahab coordinates.

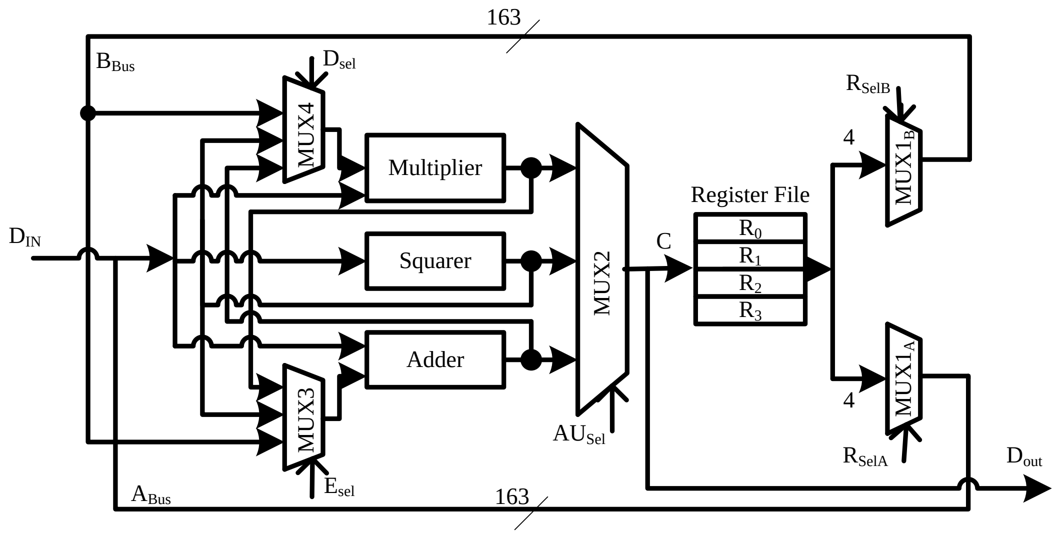


Figure 4.3: Optimized point addition/doubling unit

4.3.4 Analyzing the Design

The modifications of the López-Dahab mixed coordinates point multiplication scheme that can be obtained in Algorithm 4.3 indicate that registers R_0 and R_1 are used for storing long variables. It is assumed that field elements $Y_1, and\, Z_1$ are stored in registers R_2 and R_3 respectively; however, $X_{1(3)}$, X_2, Y_2 are defined or modified by read or write operations through BRAM. As a result, if $k_i = 1$, there will be a total of 10 read and three write accesses. However, if $k_i = 0$, the total number of read and write accesses will be three and two, respectively. Based on Figure 4.3, a forwarding path (FP) can be obtained between the finite field multiplier and adder through MUX3. The short variable resulting from the finite field multiplier is forwarded directly to the adder, and the results from the adder are then stored in the register file, as shown in step 4. Based on the developed register allocation and operation scheduling models, MUX3 can provide several FPs that can be used in different steps either from the finite field multiplier (MUL) to the adder (ADD), defined as $FP(MUL - ADD)$, or from the finite field squarer SQR to the ADD, defined as $FP(SQR - ADD)$. As mentioned in section 3.2.3, the use of storage binding as a strategy for minimizing the number of registers is evident in several steps in Algorithm 4.3. According to the variable liveness analysis of element Y_1, step 6 coalesces Y_1 by means of the intermediate variable D defined as $Y_1(D)$. On the other hand, step 24 shows the coalescing of

Y_1 by means of the intermediate variable C.

Modified register management for the left-to-right point multiplication algorithm using López-Dahab projective coordinates is presented in Algorithm 4.4. It should be noted that when the algorithm is actually implemented, additional controls are required. In the algorithm shown only the register file management is indicated. According to the new methodology presented in chapter 3, it is assumed that $R_0 - R_3$ are used to store long intermediate variables. The field elements X_1, Y_1, Z_1, X_2, Y_2, Z_2 are defined and modified via D_{IN} and D_{OUT} through the BRAM. The total number of read accesses is 30 if $k_i = 1$ and 11 if $k_i = 0$. In addition, 11 write accesses are needed if $k_i = 1$ and 4 accesses if $k_i = 0$. As shown in Figure 4.3, $FP(SQR-MUL)$ and $FP(ADD-MUL)$ can be accomplished through MUX4. The terms used and defined, such as $Z_1(T)$, $Y_1(J)$, $X_1(C)$, are examples of the applications of the coalescing operation mentioned in step 4 in the methodology explained in section 3.2.3. The freedom to schedule the movement of operand/result data using forwarding paths in the point multiplication cycles reduces the pressure on the register file access, which was one of the main motivations for the addition of the data forwarding paths.

4.4 Implementation Results and Comparison

The proposed ECP over $\mathbb{F}_{2^{163}}$ has been completely implemented for both the conventional and the optimized designs in an RTL-level VHDL [63, 90]. For a fair comparison with other architectures presented in the literature, the code was synthesized and implemented on FPGAs using Xilinx XC4VLX200. For performance analysis and low-power metrics, the Virtex-4 family was targeted. The entire design was verified on a Model-Sim SE v. 6.5e using a special test DO program. Table 4.1 summarizes the area requirements of both the conventional and the optimized ECPs and provides comparison with other related work described in [18, 19, 55, 100].

The register usage of the optimized design is 35% less than that of conventional architectures. However, using forwarding paths does indeed increase the critical path, which affects the maximum clock frequency and the total computational time for a point multiplication operation. In [18], the area used is roughly more than twice that of our proposed optimized design because the architecture in [18] is based on a seven-stage pipelined finite field multiplier. The scalar mul-

Algorithm 4.3 The modified register management for left-to-right binary point multiplication in $\mathbb{F}_{2^m}$ in López-Dahab mixed projective coordinates.

Input: P, k where $P \in E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + ax^2 + b\,,\ b \neq 0,\ k = [k_{t-1} \cdots k_1, k_0]_2$.
Output: $Q = k\,P$ where $Q = (X_3, Y_3, Z_3)$ in $\mathcal{LD}$ coordinates $\in E(\mathbb{F}_{2^m})$

```
1: for $i = t - 1$ down to 0 do
2:     $R_0 \leftarrow MUL\,(X_{1(3)},\, Z_{1(3)})$
3:     $R_1 \leftarrow SQR\,(X_{1(3)})$
4:     $X_1(C) \leftarrow ADD\,(R_1,\, Y_{1(3)})$
5:     $Y_1(D) \leftarrow MUL\,(R_0,\, X_1(C))$
6:     $Z_1(Z_3) \leftarrow SQR\,(R_0)$
7:     $R_0 \leftarrow SQR\,(X_1(C))$
8:     $R_0 \leftarrow ADD\,(R_0,\, Y_1(D))$
9:     $X_1(X_3) \leftarrow ADD\,(R_0, Z_1(Z_3))$
10:    $R_0 \leftarrow ADD\,(Z_1(Z_3),\, Y_1(D))$
11:    $R_1 \leftarrow SQR\,(R_1)$
12:    $R_1 \leftarrow MUL\,(R_1,\, Z_1(Z_3))$
13:    $FP(MUL - ADD) \leftarrow MUL\,(R_0,\, X_1(X_3))$
14:    $Y_1(Y_3) \leftarrow R_1,\, FP(MUL - ADD)$
15:    if $k_i = 1$ then
16:        $R_0 \leftarrow SQR\,(Z_1)$
17:        $FP(MUL - ADD) \leftarrow MUL(Y_2, R_0), R_0 \leftarrow ADD(Y_1, FP(MUL - ADD))$
18:        $FP(MUL - ADD) \leftarrow MUL(X_2, Z_1), R_1 \leftarrow ADD(X_1, FP(MUL - ADD))$
19:        $Y_1(C) \leftarrow MUL\,(R_1,\, Z_1)$
20:        $Z_1(Z_3) \leftarrow SQR\,(Y_1(C))$
21:        $FP(SQR - ADD) \leftarrow SQR\,(R_1)\ ,\ R_1 \leftarrow ADD(R_0,\, FP(SQR - ADD))$
22:        $R_1 \leftarrow ADD\,(R_1,\, Y_1(C))$
23:        $R_1 \leftarrow MUL\,(R_1,\, Y_1(C))$
24:        $FP(SQR - ADD) \leftarrow SQR\,(R_0)\ ,\ X_1(X_3) \leftarrow ADD(R_1,\, FP(SQR - ADD))$
25:        $R_1 \leftarrow MUL\,(R_0,\, Y_1(C))$
26:        $FP(MUL - ADD) \leftarrow MUL(X_2, Z_{1(3)}), R_0 \leftarrow ADD(X_{1(3)}), FP(MUL - ADD)$
27:        $R_0 \leftarrow ADD\,(R_1,\, Z_1(Z_3))$
28:        $Y_1(T) \leftarrow MUL\,(R_0,\, R_1)$
29:        $R_0 \leftarrow ADD\,(X_2,\, Y_2)$
30:        $R_1 \leftarrow SQR\,(Z_1(Z_3))$
31:        $FP(MUL-ADD) \leftarrow MUL(R_0, R_1), Y_1(Y_3) \leftarrow ADD(Y_1(T), FP(MUL-ADD))$
32:    end if
33: end for
34: $Return\,(Q = (X_3, Y_3, Z_3))$
```

Algorithm 4.4 The modified register management for left-to-right binary point multiplication in $\mathbb{F}_{2^m}$ in López-Dahab projective coordinates.

```
Input:  $P$, $k$ where $P \in E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + ax^2 + b\,,\ b \neq 0,\ k = [k_{t-1} \cdots k_1, k_0]_2$.
Output:  $Q = k\,P$ where $Q = (X_3, Y_3, Z_3)$ in $\mathcal{LD}$ coordinates $\in E(\mathbb{F}_{2^m})$
 1: for  $i = t-1$ downto 0  do
 2:     $R_0 \leftarrow MUL\,(X_{1(3)},\, Z_{1(3)})$
 3:     $R_1 \leftarrow SQR\,(X_{1(3)})$
 4:     $X_1(C) \leftarrow ADD\,(R_1,\, Y_{1(3)})$
 5:     $Y_1(D) \leftarrow MUL\,(R_0,\, X_1(C))$
 6:     $Z_1(Z_3) \leftarrow SQR\,(R_0)$
 7:     $R_0 \leftarrow SQR\,(X_1(C))$
 8:     $R_0 \leftarrow ADD\,(R_0,\, Y_1(D))$
 9:     $X_1(X_3) \leftarrow ADD\,(R_0, Z_1(Z_3))$
10:     $R_0 \leftarrow ADD\,(Z_1(Z_3),\, Y_1(D))$
11:     $R_1 \leftarrow SQR\,(R_1)$
12:     $R_1 \leftarrow MUL\,(R_1,\, Z_1(Z_3))$
13:     $FP(MUL - ADD) \leftarrow MUL\,(R_0,\, X_1(X_3))$
14:     $Y_1(Y_3) \leftarrow R_1,\, FP(MUL - ADD)$
15:     if $k_i = 1$  then
16:         $R_0 \leftarrow MUL\,(X_1,\, Z_2)$
17:         $X_1(B) \leftarrow MUL(X_2,\, Z_1)$
18:         $FP(SQR - MUL) \leftarrow SQR(Z_2),\ R_1 \leftarrow MUL(Y_1,\, FP(SQR - MUL))$
19:         $FP(SQR - MUL) \leftarrow SQR\,(Z_1),\ R_2 \leftarrow MUL(Y_2,\, FP(SQR - MUL))$
20:         $Y_1(I) \leftarrow ADD\,(R_0,\, R_1)$
21:         $FP(ADD - MUL) \leftarrow ADD(R_0, X_{1(B)}), Y_{1(J)} \leftarrow MUL(Y_{1(I)}, FP(ADD - MUL))$
22:         $Z_1(T) \leftarrow MUL\,(Z_1,\, Z_2)$
23:         $R_3 \leftarrow SQR\,(R_0)$
24:         $FP(SQR\,AD) \leftarrow SQR(X_1(B)), FP(AD\,MUL) \leftarrow ADD(R_3, FB(SQR\,AD))$
25:         $Z_1(Z_3) \leftarrow MUL((FP(ADD - MUL),\, Z_1(Z_3))$
26:         $FP(ADD - MUL) \leftarrow ADD\,(R_1,\, R_3), R_3 \leftarrow MUL(FP(ADD - MUL),\, X_1(B))$
27:         $FP(SQR, AD) \leftarrow SQR(X_{1(B)}), FP(AD, MUL) \leftarrow ADD(R_2, FP(SQR, AD))$
28:         $R_2 \leftarrow MUL\,(R_0,\, FP(ADD - MUL)$
29:         $X_1(X_3) \leftarrow ADD\,(R_2,\, R_3)$
30:         $R_3 \leftarrow SQR\,(X_1(X_3))$
31:         $FP(SQR - ADD) \leftarrow SQR\,(R_0),\ R_2 \leftarrow ADD(FP(SQR - ADD),\, R_3)$
32:         $R_1 \leftarrow MUL(R_1, R_2)$
33:         $R_0 \leftarrow MUL(R_0,\, Y_1(J))$
34:         $FP(ADD - MUL) \leftarrow ADD(R_0,\, R_1),\ R_2 \leftarrow MUL(FP(ADD - MUL),\, R_2)$
35:         $FP(ADD - MUL) \leftarrow ADD(Y_{1(J)}, Z_{1(3)}), R_3 \leftarrow MUL(FP(ADD - MUL), X_{1(3)})$
36:         $Y_1(Y_3) \leftarrow ADD(R_2,\, R_3)$
37:     end if
38: end for
39: $Return\,(Q = (X_3, Y_3, Z_3))$
```

Table 4.1: Comparison of the FPGA implementation of the elliptic curve scalar multiplication designs

Architecture	Technology	Total		Area	Clock Rate	
	FPGA	#FFs	#LUTs	slices	F_{MAX} MHz	CLK ns
William et al. [18]	XC4VLX200	-	-	16,209	153.90	6.497
Hyun et al. [19]	XC4VLX80	-	-	24,363	143.00	6.993
Deschamps et al. [25]	XC4VLX200	10,349	14,875	12,849	276.31	3.619
Zhang et al. [100]	XC4VLX80	-	-	20,807	185.00	5.405
Conventional (Projective)	XC4VLX200	9,303	13,352	11,327	264.84	3.776
Optimized (Projective)	XC4VLX200	5,955	9,761	7,858	169.66	5.894
Conventional (Mixed)	XC4VLX200	8,657	12,660	10,658	263.62	3.793
Optimized (Mixed)	XC4VLX200	5,563	9,415	7,490	168.63	5.930

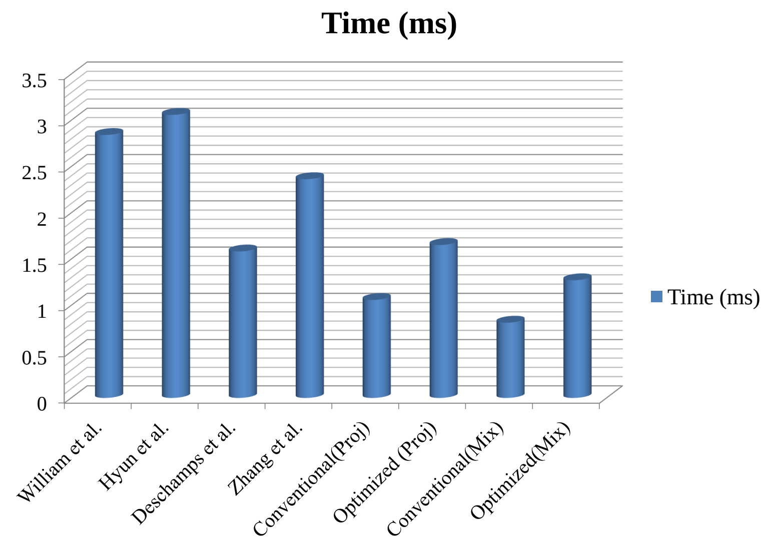


Figure 4.4: Comparison on the point multiplication computational times of the ECP designs

tiplication design using affine point representation that was implemented in [25] has been synthesized and simulated for the current work using the same FPGA device used for the conventional and optimized designs tested in this study. The superior results produced by the conventional architecture compared to the one implemented by [25] is due largely to the use of projective coordinates. It is particularly interesting to note that in [19] the maximum frequency is even lower than in [18], as shown in Figure 4.4, because in the new design, the critical path is approximately equal to the time delay of one iteration finite field multiplier plus the adder. However, in [19] the critical path is a 55-bit-sized multiplier plus the squarer and the adder. The possibility exists that the conventional and the optimized design can fit into the XC4VLX80 device because the number of available slices is 35,840. The developed design therefore requires less area than the architecture presented in [19]. Using three finite field multi-core elliptic curve processors significantly decreases the computational time for a point multiplication. On the other hand, it uses approximately three times more hardware resources than the developed mixed coordinate optimized design.

To sum up, a comparison of the conventional and the optimized ECPs with

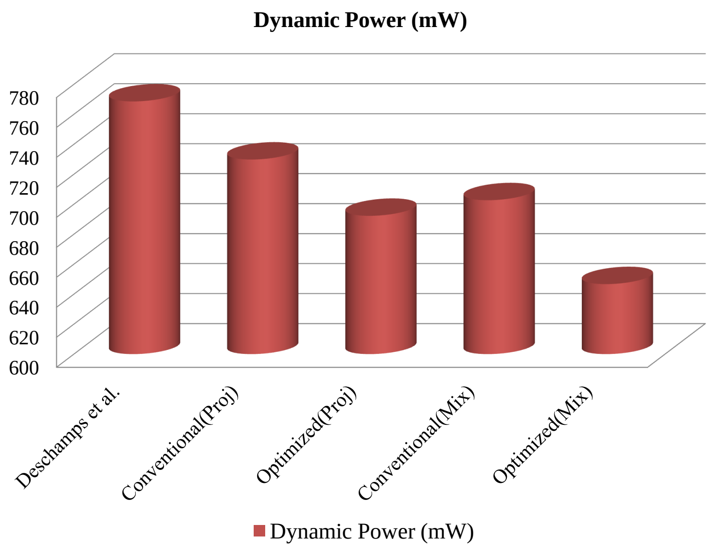


Figure 4.5: Dynamic power comparisons of the ECP designs

respect to the dynamic power dissipated in the point multiplication unit is shown in Figure 4.5. The dynamic power consumption was estimated using a Xilinx ISE X power analyzer. Based on the implementation given in [25], the power consumed by their design was estimated. The table provided indicated a gradual decrease of about 15.8% in the total dynamic power dissipation between the design in [25] and the proposed mixed coordinate optimized design. However, the difference in power consumption between the conventional and the optimized designs is only approximately 8%. The small reduction in the total power results from the large amount of power dissipated specifically inside the multiplier, which is an indication that future work needs to address the development of a methodology for optimizing power dissipation in the finite field multiplier. There is also a possibility that reducing the area requirements may lead to a significant reduction in total power consumption by the optimized design because it would fit into a smaller FPGA device. This improvement would clearly result an additional reduction, especially in power leakage.

4.5 Conclusion

The optimization of area and power are two important design issues with respect to the ECC used in many embedded systems. One benefit of ECC is that it requires a much shorter key length than other public key cryptosystems in order to provide an equivalent level of security. However, the hardware implementation of elliptic curve processor (ECP) for lightweight devices is a challenge. In this work, an efficient processor is proposed for ECC that aims to reduce the number of registers relative to those that have been presented in the literature. Forwarding paths in the ECP were employed as a means of avoiding the writing/reading of short variables to/from the register file. The proposed ECP design was implemented over $\mathbb{F}_{2^{163}}$ on a Xilinx XC4VLX200 FPGA device in order to verify its functionality and to measure its performance. The results of this work show a saving in area of up to 38% for the number of Flip Flops (#FF) and up to 27% with respect to the number of look-up tables (#LUTs). The performance overhead is equal to $1.8\ ns$ to be added on the critical path ECP. When constrained devices must be used for implementing a public key cryptographic system, an ECP implementation with a smaller key size can be employed. This study has demonstrated the small-area implementation of an ECP for both projective and mixed coordinates. Using the proposed model of register allocation and operation scheduling, area saving of approximately 38% have been demonstrated at the point arithmetic level. The ECP has been implemented using FPGAs. An improved register management scheme has been developed for point multiplication using projective and mixed López-Dahab coordinates as well as forwarding paths that lead to reduction in area and in dynamic power. Point multiplication computational times of $1642.88\ \mu s$ and $1271.99\ \mu s$ have been achieved for projective and mixed coordinates, respectively. These results show that optimizing the number of write backs into register file, analyzing the data dependency of intermediate variables and scheduling scalar multiplication operations must be carefully handled in order to save both area and power.

Chapter 5

Efficient Implementation of Genus 2 Hyperelliptic Curve Algorithms

5.1 Introduction

Hyperelliptic Curve Cryptography (HECC) was proposed in 1989 by Koblitz [48] and can be seen as a generalization of elliptic curve cryptography (ECC). The main advantage of HECC is that it provides the same level of security as ECC, but the ground field is only half the size. For example, a hyperelliptic curve of genus 2 over $\mathbb{F}_{2^{83}}$ can provide the same level of security as an elliptic curve defined over $\mathbb{F}_{2^{163}}$, and such shorter operands appear promising for applications in constrained environments. While ECC applications are highly developed in practice, the use of HECC is still of only academic. However, the transformation in a group of points over elliptic curve cryptography (ECC) is accepted as a modern public key primitive [4]. Recent published works, such as [79] have shown that the transformation in the Jacobian of HECC is considered the most promising substitution in ECC and that the performance of HECC can be compared to that of ECC. The cryptographic transformation in the Jacobian is also based on the scalar multiplication [48] of reduced divisors, called divisor multiplication for the purposes of this study.

The main emphasis of this chapter is the application of architecture synthesis and optimization techniques to algorithms with the goal of developing efficient explicit formulas over a finite field of even characteristic. Efficient explicit formulas are presented for a variety of inversion-free coordinate systems: projective, new weighted, and recent coordinates. These three systems enable the avoid-

ance of inversions in the group operation. To reduce the number of registers, architectural synthesis techniques were applied, which include consideration of the greatest power consumer in the HECC processor. Architecture optimization techniques were next applied in order to analyze the overall tradeoffs between area, power consumption, and computation time. A brief overview of previous work related to the implementation of HECC processor hardware is included. This work also describes the hardware architecture utilized for HECC for the three inversion-free coordinates and presents an efficient architecture for implementing divisor addition, divisor doubling, and the calculation of the divisor multiplication. Our goal was to determine the lower limits of an area/power public key processor for HECC curves. To this end, predictability was sacrificed in order to design inversion-free coordinates for characteristic 2 curves, which is quite reasonable for constrained devices. All of the proposed architectures are both area and power optimized. Based on the author's previous implementation on ECC processors, the memory and register requirements for storing points and temporary variables can contribute substantially (more than 58%) to the overall size of an ECC processor. The goal of this study was hence an efficient architecture that requires less memory and fewer register requirements even if it entails a small computational disadvantage. To the best of the author's knowledge, this work is the first to use synthesis and optimization to develop an efficient hardware architecture for projective, new weighted, and recent coordinates over a binary field.

5.2 Previous Work related to HECC Hardware Implementation

Relatively few HECC hardware implementations have been reported e.g., [45, 94, 93]. The advantages of hardware implementations include better performance and increased physical security than with software solutions. This section provides a brief summary of previous attempts to use hardware to implement HECC processor for performing divisor multiplication. The first work that proposed an architecture for the hardware implementation of a hyperelliptic cryptosystem was performed by Thomas Wollinger in 2001 [92]. The implementation was based on consideration of the general form of Cantor's algorithm with its underlying polynomial arithmetic. The implemented hyperelliptic curve of genus 2 is

Table 5.1: Estimated timing results of previous HECC group operations and divisor multiplications

Reference	Field	G	Divisor			Frequency
			ADD	DBL	MUL	F_{max} MHz
[92]	$\mathbb{F}_{2^{81}}$	1	$118\mu s$	$71\mu s$	$25ms$	54
[20]	$\mathbb{F}_{2^{83}}$	1	$71\mu s$	$62\mu s$	$10ms$	45
		4	$52\mu s$	$43\mu s$	$9ms$	
[15]	$\mathbb{F}_{2^{113}}$	1	$105\mu s$	$90\mu s$	$19ms$	45
[21]	$\mathbb{F}_{2^{163}}$	1	$147\mu s$	$123\mu s$	$40ms$	45
		4	$109\mu s$	$85\mu s$	$35ms$	
[29]	$\mathbb{F}_{2^{113}}$	1	$35.8\mu s$	$30.7\mu s$	$7.53ms$	45.6
		4	$10\mu s$	$8.58\mu s$	$2.12ms$	46.7
Type 1 in [45]	$\mathbb{F}_{2^{83}}$	32	n.a.	n.a.	$436\mu s$	62.9
Type 2 in [45]					$791\mu s$	50.1
Type 3 in [45]					$1\,020\mu s$	50.5

given by $C : y^2 + xy = x^5 + f_7x^7 + f_3x^3 + 1$. The author developed computer architecture for the appropriate algorithms required for implementing the necessary field and polynomial operations in the HECC hardware. The architectures were developed for a reconfigurable platform based on field programmable gate arrays (FPGAs).

The first complete hardware implementation of an HECC processor, however, was presented in [15]. The final processor developed in this study included two polynomial multipliers and one each of the other polynomial computation blocks, including the adder, the divisor, and the squarer. It was also based on Cantor's algorithm but included an alternative method derived from the calculation of the greatest common divisor (GCD). Boston *et al.* [15] provided actual time measurements on real hardware and presented concrete performance results from a hardware-based genus two hyperelliptic curve coprocessor over $\mathbb{F}_{2^{113}}$. The hardware implementation was conducted on a Xilinx Virtex II FPGA. The authors used Verilog HDL and the Xilinx Integrated Software Environment to synthesize and implement the logic design. The implementation was based on consideration of the following curve: $y^2 + xy = x^5 + f_2x^2 + 1$, i.e., all coefficients are elements of $\mathbb{F}_2$. Although these type of curves are Koblitz curves, the authors made no use of the Frobenius automorphism. Reference [58] provides additional details about the Forbenius automorphism.

In [20, 21] the authors presented extended results for the work reported in

[15]. They implemented an HECC coprocessor using a variety of base fields ranging from $\mathbb{F}_{2^{83}}$ to $\mathbb{F}_{2^{163}}$ in two different digit sized multipliers: G $= 1$ bit and G $= 4$ bits. The divisor multiplication took between $9\ ms$ and $40\ ms$ and used between $22\,000$ and $118\,000$ slices. Based on the implementation numbers listed in Table 5.2, it seems that the design with $118\,000$ slices involves unreasonable hardware requirements because the maximum available number of slices in a Xilinx Vertix 4, for example, is approximately $90\,000$.

In [45], the authors implemented three different designs of an HECC processor, ranging from high-performance to moderate-area designs. They chose the following parameters for their implementation: underlying field $\mathbb{F}_{2^{89}}$, curve parameters $h(x) = x$ and $F(x) = x^5 + f_1x + f_0$, and explicit formulas based on affine coordinates [61]. In the case of the high-performance design which is Type 1, two independent arithmetic units are used: one for group addition and one for group doubling. In the Type 2 design, the HECC processor provides only one arithmetic unit shared through group addition and group doubling. The final design presented [45], Type 3, aimed for a low area. In this type of design, the authors used only memory rather than both memory and registers, claiming that the decoding logic for reading/writing data from/to the register file is intrinsically implemented inside the memory, which reduces the total design area. However, the total number of clock cycles for the overall computation increases because of the expensive movement of data from and to the memory. Reference [29] reported the first implementation of a HECC processor on an FPGA based on the explicit formulas that use projective coordinates, and the implementation was also over $\mathbb{F}_{2^{113}}$.

The timing and area results for all of these studies are shown in Table 5.1 and Table 5.2, respectively. The results that are achievable through a projective explicit formula that is based on Harley's algorithms are far greater than these that can be achieved via Cantor's algorithm because the four-level hierarchy based on Cantor's algorithm, that is, finite field arithmetic, polynomial ring arithmetic, computations on the Jacobian of the curve and divisor multiplications, is reduced to a three-level hierarchy. Based on Lange's explicit formula for projective and mixed coordinates, the three levels of such a hierarchy are finite field arithmetic, computations of divisor addition and doubling, and divisor multiplication. In addition to the studies listed in Table 5.1 and Table 5.2, the tables includes reports of some ASIC implementations of HECC using projective

Table 5.2: Estimated area results in slices of previous HECC hardware implementation

Reference	Technology	Field	G	Divisor		
				ADD	DBL	MUL
[20]	Xilinx Virtex II	$\mathbb{F}_{2^{83}}$	1	10 400	10 200	22 000
			4	29 700	29 300	60 000
[15]	Virtex II 2VP30	$\mathbb{F}_{2^{113}}$	1	16 600	15 100	30 816
[21]	Xilinx Virtex II	$\mathbb{F}_{2^{163}}$	1	20 400	20 100	42 000
			4	58 400	57 600	118 000
[29]	XC2V8000	$\mathbb{F}_{2^{113}}$	1	9514	9052	22 183
			4	10 988	10 087	25 911
Type 1 in [45]	XC2V4000	$\mathbb{F}_{2^{83}}$	32	n.a.	n.a.	9950
Type 2 in [45]				n.a.	n.a.	7096
Type 3 in [45]				n.a.	n.a.	4995

coordinates. For example, in [84], the author proposed an HECC processor using 130 nm CMOS technology. The processor runs at 500 MHz and can perform one divisor multiplication of HECC over $\mathbb{F}_{2^{83}}$ in 63 μs.

5.3 Explicit Formulas on Genus 2 Curves over a Binary Field

The first attempt to avoid using Cantor's algorithm to increase the speed of group operations in the Jacobian on hyperelliptic curves of genus 2 was made by Robert Harley in 2000 [33]. He describes an efficient algorithm, carefully optimized to reduce the number of required group operations. For simplicity, Harley restricted his algorithm to odd characteristics, with all tricks able to be carried to even characteristics. His basic concept was to explicitly compute the divisor addition and divisor doubling algorithms using Cantor's algorithm so that only field operations and no polynomial operations are necessary. Further simplifications are obtained through the use of the Chinese remainder theorem in the group addition algorithm and through the simplification of the group doubling algorithm with the help of one Newton's iteration. The latest improvement of genus 2 HECC of odd characteristic was developed by [88]. The authors used Montgomery's trick of simultaneous inversions to compute two inverses by performing only one field inversion and three field multiplications. The extension of the explicit formulas developed by [88] for arithmetic on genus two curves to

fields of even characteristic and to arbitrary equations of the curve was achieved by Tanja Lange [59]. Lange also presented timings for the implementation of the formulas using a variety of libraries for the field arithmetic and determined the exact number of operations needed for performing the addition and doubling required in the most common cases.

An analysis of the complexity of the operations using the known methods of arithmetic transforms in Jacobian genus 2 HECC over even and odd characteristics demonstrates that the existing methods are already efficient but that room is available for further refinement. Based on the classification of genus 2 curves in even characteristics, the complexity of operations on recent coordinates for Type 1 designs was determined using an assumed count of $h_2 = 1$ and $f_4 = f_3 = f_2 = 0$. In this study, efficient method of hardware implementation for a genus two Harley algorithm over $\mathbb{F}_{2^m}$ is presented. The new method is based on several active steps of architecture synthesis and optimization, including register allocation, operation scheduling, storage binding, and data path/control unit analysis. A comprehensive hardware implementation for inversion-free hyperelliptic curve algorithms has been demonstrated.

5.3.1 Cantor's Algorithms for Explicit Formula over Even Characteristics

For general cases and computation purposes, a group operation is based on Cantor's algorithm [16], which operates directly in Mumford's representation [74]. Cantor's original version worked only in odd characteristics and was extended for all fields by Koblitz [48]. Chapter 2 gives the algorithms restricted to curves over fields of characteristic two. To optimize the execution time of Cantor's algorithm, one can write the steps for adding, doubling, and reducing divisors explicitly, i.e., calculate the resulting divisor without the use of polynomial arithmetic. This method results in a formula similar to those known for elliptic curves. The concept behind explicit formula is to replace the polynomial-based form of Cantor's algorithm with a coefficient-based approach. These formulas are dependent on whether the divisors are distinct (group addition) or equal (group doubling). Using explicit formula has a number of advantages [8] that result in a significant reduction in the area required for and the speed of the computation:

- In Cantor's algorithm, some of the partial computations may be performed twice, with the only difference being the names of the variables. In an explicit formula, these duplications are avoided because those intermediate values are held in the memory or in instant registers.

- The number of multiplications can be reduced [33] with the use of the Karatsuba multiplication algorithm [42], the Chinese remainder theorem, and a Newton iteration. The resulting explicit formulas are advantageous in applications where a short computation time is critical.

- The goal of introducing inversion-free formula [57] for calculating a group operation on a genus 2 HECC can be achieved through the addition of another coordinate to represent the elements of the divisor class group. The resulting explicit formula are advantageous in applications where inversions are more expensive than multiplication.

- More efficient inversion-free explicit formula have been developed for projective and weighted coordinates on genus 2 for even characteristics when $h_2 \in \{0, 1\}$ and for recently used coordinates when $h_2 = 0$. In addition, [22] has provided a thorough comparison of arithmetic on the hyperellitpic curves of genus 2 curves that contains various coordinates.

Most of the previous work has attempted to optimize explicit formula for hyperelliptic curve odd characteristics or to optimize the execution time of odd characteristics. This study takes a slightly different approach, which optimizes the number of registers required for storing intermediate values. In [72], related work was conducted, with the goal of minimizing the memory requirement. After the completion of the register allocation via variable liveness analysis, operation scheduling via forwarding paths, and storage binding via efficient register binding steps, the final efficient explicit formulas are determined for different inversion-free coordinates. Table 5.3 shows the amount of register complexity when various inversion-free coordinate systems of divisor addition and divisor doubling are optimized using the three optimizing techniques presented in Chapter 3. The third column indicates the number of registers for new weighted coordinates obtained by the method reported in [72]. The fourth and the last columns show the number of registers obtained using the modified register allocation and operation scheduling and storage binding algorithms described in Chapter 3. The following subsenction introduces explicit formula for recent coordinates in even

Table 5.3: Comparison of the register requirements for a variety of coordinates systems of divisor addition and divisor doubling

Inversion-free Systems	References	[72]	This Work	
		Register Reuse	Allocation	Scheduling & Binding
Projective Coordinates ($\mathcal{P}$)				
$\mathcal{P}_{DBL_{h_2 \neq 0}}$	[22]	n. a.	17	14
$\mathcal{P}_{DBL_{h_2 = 0}}$	[22]	n. a.	13	10
$\mathcal{P}_{ADD_{h_2 \neq 0}}$	[61]	n. a.	23	17
$\mathcal{P}_{ADD_{h_2 = 0}}$	[93]	n. a.	21	15
$\mathcal{P}_{mADD_{h_2 \neq 0}}$	[61]	n. a.	23	17
$\mathcal{P}_{mADD_{h_2 = 0}}$	[50]	n. a.	19	13
New Weighted Coordinates ($\mathcal{N}$)				
$\mathcal{N}_{DBL_{h_2 \neq 0}}$	[22]	20	18	14
$\mathcal{N}_{DBL_{h_2 = 0}}$	[61]	16	15	11
$\mathcal{N}_{ADD_{h_2 \neq 0}}$	[22]	23	23	18
$\mathcal{N}_{ADD_{h_2 = 0}}$	[61]	23	22	17
$\mathcal{N}_{mADD_{h_2 \neq 0}}$	[61]	20	22	17
$\mathcal{N}_{mADD_{h_2 = 0}}$	[27]	19	21	16
Recent Coordinates ($\mathcal{R}$)				
$\mathcal{R}_{DBL_{h_2 \neq 0}}$	This Work (section 5.4)	n. a.	17	13
$\mathcal{R}_{DBL_{h_2 = 0}}$	[22]	n. a.	11	10
$\mathcal{R}_{ADD_{h \neq 0}}$	This Work (section 5.4)	n. a.	24	18
$\mathcal{R}_{ADD_{h_2 = 0}}$	[22]	n. a.	23	16
$\mathcal{R}_{mADD_{h_2 \neq 0}}$	This Work (section 5.4)	n. a.	22	15
$\mathcal{R}_{mADD_{h_2 = 0}}$	[22]	n. a.	19	13

characteristic when $h_2 \neq 0$ based on coordinates conversion from *new weighted* coordinates in section 14.5 in [22] to *recent* coordinates.

5.3.2 Recent Coordinates in Even Characteristic When $h_2 \neq 0$

This section presents the algorithms for computing with a recent coordinate system in even characteristic when $h_2 \neq 0$. The more uncommon case of $h_2 = 0$ was considered in [22]. For the purpose of the current study, it was assumed that $h_2 \neq 0$. With this approach, the algorithms were developed from Algorithm 14.47 and Algorithm 14.48 in [22] for new weighted coordinates in even characteristic. New weighted coordinates $\mathcal{N}$ were presented by Lange [60]. For the general case

in even characteristic, it is most useful to use a set of coordinates extended by means of precomputations. Let $\mathcal{N}$ denote $[U_1, U_0, V_1, V_0, Z_1, Z_2, z_1, z_2, z_3, z_4]$ with the interpretation $u_i = U_i/Z_1^2$, $v_i = V_i/(Z_1^3 Z_2)$ and the precomputations $z_1 = Z_1^2$, $z_2 = Z_2^2$, $z_3 = Z_1 Z_2$ and $z_4 = z_1 z_3$. Let R denote $[U_1, U_0, V_1, V_0, Z, z]$ with $u_i = U_i/Z$, $v_i = V_i/Z^2$ and the precomputations $z = Z^2$. In Algorithms 5.1, 5.2, and 5.3 the intermediate steps are listed for divisor doubling, addition, and mixed addition for recent coordinates in even characteristics together with the number of multiplications (MUL), squaring (SQR), and additions (ADD) needed. Because it is assumed that $h_2, h_1, f_4 \in \{0, 1\}$, operations with these coefficients are not counted. Explicit formulas are derived simply based on the assumed method presented in [22] for converting new weighted coordinates into recent coordinates and using the explicit formula for new weighted coordinates for even characteristic when $h_2 \neq 0$. For example, the following steps show the development of doubling in recent coordinates in even characteristics when $h_2 \neq 0$:

1. In step 1, the calculation of Z_2' is skipped, if $Z_1 = Z_2$, $z_1 = Z$, $z_4 = Z^2$, with$u_i = U_i/Z$, $v_i = V_i/Z^2$ is assumed.

2. Step 2 requires no change in computation almost inverse which is considered as a renaming of two intermediate variables $\tilde{V_1}$ and w_3.

3. The operation count of step 3 is reduced to $3\,MUL$ because two field multiplications of $t_1 \leftarrow w_3 z_2 + V_1 h_2 z_3$ are saved through the use of $t_1 \leftarrow Z(w_3 + V_1 h_2)$.

4. In step 4, the computation of $s_0 \leftarrow w_0 + U_0 w_1 z_1$ is replaced with $s_0 \leftarrow w_0 + U_0 w_1 Z$.

5. In step 5, the calculation of z_1', z_2', z_3' and z_4' is skipped, if $Z' \leftarrow s_1 Z$, $z' \leftarrow Z'^2$ is assumed.

6. In step 6, if $z_3' = Z'$ is assumed, the computation of $l_2 \leftarrow l_2 + S + h_2 z_3'$ is replaced with $l_2 \leftarrow l_2 + S + h_2 Z'$.

For steps 7, 8 and 9, $U_0' \leftarrow S_0 + Z'y$, $U_1' \leftarrow h_2 Z'$, $V_1' \leftarrow w_1 + Z'(l_1 + RV_1 + U_0') + z'h_1$, and $V_0' \leftarrow w_0 + Z'(l_0 + RV_0) + z'h_0$. Explicit formulas are derived in a manner similar to those for computing divisor addition and divisor mixed addition and are presented in Algorithms 5.2, and 5.3, respectively.

Algorithm 5.1 Doubling in recent coordinates $g = 2$, $h_2 \neq 0$, in even characteristic

Input: $D = [U_1, U_0, V_1, V_0, Z, z]$

Output: $[U_1', U_0', V_1', V_0', Z', z'] = [2]D$

```
1. compute resultant and precomputations                 [$8MUL + 3SQR + 7ADD$]
   $\tilde{h}_1 \leftarrow Z\,h_1$ and $\tilde{h}_0 \leftarrow Z\,h_0$
   $\tilde{V}_1 \leftarrow \tilde{h}_1 + h_2U_1$ and $\tilde{V}_0 \leftarrow \tilde{h}_0 + h_2U_0$
   $w_0 \leftarrow V_1^2$, $w_1 \leftarrow U_1^2$ and $w_2 \leftarrow \tilde{h}_1^2 + h_2^2w_1$
   $w_3 \leftarrow Z\,(h_1U_1 + h_2U_0 + \tilde{h}_0) + h_2w_1$
   $r \leftarrow w_2U_0 + \tilde{V}_0w_3$, $\tilde{Z} \leftarrow Z\,r$
2. compute almost inverse
   $\mathrm{inv}_1 \leftarrow \tilde{V}_1$ and $\mathrm{inv}_0 \leftarrow w_3$
3. compute $t$                                            [$3MUL + 6ADD$]
   $w_3 \leftarrow f_3Z^2 + w_1$ and $t_1 \leftarrow Z(w_3 + V_1h_2)$
   $t_0 \leftarrow U_1t_1 + w_0 + z\,(V_1h_1 + V_0h_2 + f_2z)$
4. compute $s = (t\,\mathrm{inv}) \bmod u$                [$6MUL + 6ADD$]
   $w_0 \leftarrow t_0\,\mathrm{inv}_0$ and $w_1 \leftarrow t_1\,\mathrm{inv}_1$
   $s_1 \leftarrow (\mathrm{inv}_0 + \mathrm{inv}_1)(t_0 + t_1) + w_0 + w_1(1 + U_1)$
   $s_0 \leftarrow w_0 + U_0w_1Z$
5. precomputations                                        [$6MUL + 2SQR + 2ADD$]
   $y \leftarrow h_2s_0 + s_1(h_2U_1 + \tilde{h}_1)$, $Z' \leftarrow s_1Z$, $S_0 \leftarrow s_0^2$ and $s_1 \leftarrow Z's_1$
   $S \leftarrow s_0Z'$, $R \leftarrow \tilde{Z}Z'$, $s_0 \leftarrow s_0s_1$ and $z' \leftarrow Z'^2$
6. compute $l$                                            [$3MUL + 6ADD$]
   $l_2 \leftarrow s_1U_1$, $l_0 \leftarrow s_0U_0$ and $l_1 \leftarrow (s_1 + s_0)(U_1 + U_0) + l_0 + l_2$
   $l_2 \leftarrow l_2 + S + h_2Z'$
7. compute $U'$                                           [$1MUL + 1ADD$]
   $U_0' \leftarrow S_0 + Z'y$ and $U_1' \leftarrow h_2Z'$
8. precomputations                                        [$2MUL + 1ADD$]
   $l_2 \leftarrow l_2 + U_1'$, $w_0 \leftarrow l_2U_0'$ and $w_1 \leftarrow l_2U_1'$
9. compute $V'$                                           [$6MUL + 7ADD$]
   $V_1' \leftarrow w_1 + Z'(l_1 + RV_1 + U_0') + z'h_1$
   $V_0' \leftarrow w_0 + Z'(l_0 + RV_0) + z'h_0$
10. return $[U_1', U_0', V_1', V_0', Z', z')$     [total complexity: $35MUL + 5SQR + 36ADD$]
```

Algorithm 5.2 Addition in recent coordinates $g = 2$, $h_2 \neq 0$, in even characteristic

Input: Two divisor classes D_1 and D_2 represented by $D_1 = [U_{11}, U_{10}, V_{11}, V_{10}, Z_1, z_1]$ and $D_2 = [U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$.

Output: The divisor class $[U'_1, U'_0, V'_1, V'_0, Z', z'] = D_1 \oplus D_2$.

```
1. precomputations                                                   [$5MUL$]
   $\tilde{U}_{21} \leftarrow U_{21}Z_1$, $\tilde{U}_{20} \leftarrow U_{20}Z_1$, $\tilde{V}_{21} \leftarrow V_{21}z_1$ and $\tilde{V}_{20} \leftarrow V_{20}z_1$
   $\tilde{Z}_1 \leftarrow Z_1Z_2$
2. compute resultant $r = \mathrm{Res}(U_1, U_2)$                  [$8MUL + 1SQR + 4ADD$]
   $y_1 \leftarrow U_{11}Z_2 + \tilde{U}_{21}$, $y_2 \leftarrow U_{10}Z_2 + \tilde{U}_{20}$ and $y_3 \leftarrow U_{11}y_1 + y_2Z_1$
   $r \leftarrow y_2y_3 + y_1^2U_{10}$, $\tilde{Z}_2 \leftarrow r\tilde{Z}_1$and $Z' \leftarrow \tilde{Z}_2\tilde{Z}_1$
3. compute almost inverse of $u_2$ modulo $u_1$
   $\mathrm{inv}_1 \leftarrow y_1$ and $\mathrm{inv}_0 \leftarrow y_3$
4. compute $s$                                                       [$8MUL + 8ADD$]
   $w_0 \leftarrow V_{10}z_2 + \tilde{V}_{20}$, $w_1 \leftarrow V_{11}z_2 + \tilde{V}_{21}$, $w_2 \leftarrow \mathrm{inv}_0w_0$ and $w_3 \leftarrow \mathrm{inv}_1w_1$
   $s_1 \leftarrow (\mathrm{inv}_0 + Z_1\mathrm{inv}_1)(w_0 + w_1) + w_2 + w_3(Z_1 + U_{11})$
   $s_0 \leftarrow w_2 + U_{10}w_3$
5. precomputations                                                   [$8MUL + 2SQR + 1ADD$]
   $\tilde{s}_0 \leftarrow s_0\tilde{Z}_1$, $S_0 \leftarrow \tilde{s}_0^2$, $Z' \leftarrow s_1\tilde{Z}_1$ and $R \leftarrow rZ'$
   $y_4 \leftarrow s_1(y_1 + \tilde{U}_{21})$, $U'_1 \leftarrow y_1s_1$, $s_1 \leftarrow s_1Z'$ and $s_0 \leftarrow s_0Z'$
   $z' \leftarrow Z'^2$ and $\tilde{h}_1 \leftarrow h_1Z'$
6. compute $l$                                                       [$3MUL + 2ADD$]
   $l_2 \leftarrow s_1\tilde{U}_{21}$, $l_0 \leftarrow s_0\tilde{U}_{20}$ and $l_1 \leftarrow (s_0 + s_1)(\tilde{U}_{20} + \tilde{U}_{21}) + l_0 + l_2$
7. compute $U'$                                                      [$5MUL + 7ADD$]
   $U'_0 \leftarrow S_0 + y_4U'_1 + y_2s_1 + Z'(h_2(\tilde{s}_0 + y_4) + y_1\tilde{Z}_1) + \tilde{h}_1$
   $U'_1 \leftarrow Z'(U'_1 + h_2)$
8. precomputations                                                   [$3MUL + 3ADD$]
   $l_2 \leftarrow l_2 + Z'(\tilde{s}_0 + h_2) + U'_1$, $w_0 \leftarrow l_2U'_0$ and $w_1 \leftarrow l_2U'_1$
9. compute $V'$                                                      [$5MUL + 7ADD$]
   $V'_1 \leftarrow w_1 + Z'(l_1 + R\tilde{V}_{21} + U'_0 + \tilde{h}_1)$
   $V'_0 \leftarrow w_0 + Z'(l_0 + R\tilde{V}_{20}) + z'h_0$
10. return $[U'_1, U'_0, V'_1, V'_0, Z', z')$      [total complexity: $45MUL + 3SQR + 32ADD$]
```

Algorithm 5.3 Mixed addition in recent coordinates $g = 2$, $h_2 \neq 0$, in even characteristic

Input: Two divisor classes D_1 and D_2 represented by $D_1 = [U_{11}, U_{10}, V_{11}, V_{10}]$ and $D_2 = [U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$.

Output: The divisor class $[U_1', U_0', V_1', V_0', Z', z'] = D_1 \oplus D_2$.

```
1. compute resultant $r = \mathrm{Res}(U_1, U_2)$                    $[6MUL + 2SQR + 4ADD]$
   $y_1 \leftarrow U_{11}Z_2 + U_{21}$, $y_2 \leftarrow U_{10}Z_2 + U_{20}$ and $y_3 \leftarrow U_{11}y_1 + y_2$
   $r \leftarrow y_2y_3 + y_1^2U_{10}$, $\tilde{Z}_2 \leftarrow rZ_2^2$ and $Z' \leftarrow \tilde{Z}_2$
2. compute almost inverse of $u_2$ modulo $u_1$
   $\mathrm{inv}_1 \leftarrow y_1$ and $\mathrm{inv}_0 \leftarrow y_3$
3. compute $s$                                                      $[7MUL + 7ADD]$
   $w_0 \leftarrow V_{10}z_2 + V_{20}$, $w_1 \leftarrow V_{11}z_2 + V_{21}$, $w_2 \leftarrow \mathrm{inv}_0w_0$ and $w_3 \leftarrow \mathrm{inv}_1w_1$
   $s_1 \leftarrow (\mathrm{inv}_0 + \mathrm{inv}_1)(w_0 + w_1) + w_2 + w_3(1 + U_{11})$
   $s_0 \leftarrow w_2 + U_{10}w_3$
4. precomputations                                                  $[8MUL + 2SQR + 1ADD]$
   $\tilde{s}_0 \leftarrow s_0Z_2$, $S_0 \leftarrow \tilde{s}_0^2$, $Z' \leftarrow s_1Z_2$ and $R \leftarrow rZ'$
   $y_4 \leftarrow s_1(y_1 + U_{21})$, $U_1' \leftarrow y_1s_1$, $s_1 \leftarrow s_1Z'$ and $s_0 \leftarrow s_0Z'$
   $z' \leftarrow Z'^2$ and $\tilde{h}_1 \leftarrow h_1Z'$
5. compute $l$                                                      $[3MUL + 4ADD]$
   $l_2 \leftarrow s_1U_{21}$, $l_0 \leftarrow s_0U_{20}$ and $l_1 \leftarrow (s_0 + s_1)(U_{20} + U_{21}) + l_0 + l_2$
6. compute $U'$                                                     $[4MUL + 6ADD]$
   $U_0' \leftarrow S_0 + y_4U_1' + y_2s_1 + Z'(h_2(\tilde{s}_0 + y_4) + y_1\tilde{Z}_2) + h_1$
   $U_1' \leftarrow Z'(U_1' + h_2)$
7. precomputations                                                  $[3MUL + 2ADD]$
   $l_2 \leftarrow l_2 + Z'(\tilde{s}_0 + h_2) + U_1'$, $w_0 \leftarrow l_2U_0'$ and $w_1 \leftarrow l_2U_1'$
8. compute $V'$                                                     $[5MUL + 7ADD]$
   $V_1' \leftarrow w_1 + Z'(l_1 + RV_{21} + U_0' + h_1)$
   $V_0' \leftarrow w_0 + Z'(l_0 + RV_{20}) + z'h_0$
9. return $[U_1', U_0', V_1', V_0', Z', z')$        [total complexity: $36MUL + 4SQR + 31ADD$]
```

5.4 Efficient Explicit Formula of Genus 2 over Binary Field

For HECC, two types of finite fields are currently under consideration: binary and prime fields. A binary field offers far more options because of there are many choices for bases, irreducible polynomials, composite fields, etc. Explicit formulas for curves of genus two have been studied extensively [38, 41, 80, 61], both for binary and prime fields. The work in this thesis is an extensive study of the challenge related to efficient hardware implementation of genus two hyperelliptic Jacobians over a binary field for a variety of inversion-free coordinates.

First, a detailed explicit formula was presented for both doubling and addition in recent coordinates for even characteristics in the case of $h_2 \neq 0$ as has been shown in section 5.3.2. Methods of improving the best-known explicit formula in projective, new weighted, and recent coordinates for genus two curves are explained in Chapter 3. Special models for register allocation via variable liveness analysis, operation scheduling via forwarding paths, and storage binding via efficient register spilling were used as appropriate in order to reformulate the explicit formula. The goals were to minimize the area and power and to moderate the computational time penalties associated with architecture synthesis and optimization problems, which have a greater impact as the field and group sizes increase.

For the general case in even characteristic and $h_2 \neq 0$, it was assumed that $f_4 = f_3 = f_2 = 0$ and $h_2 = 1$. The algorithm included h_2, f_3, and f_2 in the algorithm, but there values were not included in the operations counting, and f_4 was completely omitted. It should be noted that different choices for f_4, f_3, and f_2 give rise to different isomorphism classes while $h(x)$ distinguishes only between the two quadratic twists [22]. Sections A.1, A.2, A.3 contain the efficient register management formula for *new weighted*, *projective*, and *recent* coordinates in even characteristic, respectively. The algorithms included in these sections indicate the register transfer statements at each step. For example, Algorithm 5.4 shows the modified register management of the divisor doubling for *recent* coordinates in even characteristics when $h_2 = 0$. explained in chapter 3, the algorithm shows that the design is implemented using ten registers $R_0, R_1, R_2, \ldots, R_9$. In addition, at the beginning of the divisor doubling operation, it is assumed that four read accesses from the memory are performed in order to store U_1, U_0, V_1, V_0 and Z in

R_0, R_1, R_2, R_3 and R_4, respectively. At each step of the register file management, each field operation is performed according to the explicit formula as developed in [22]. To improve readability, Table 5.4 and 5.5 present the application of RAVLA strategy that has been composed by this work in Chapter 3 and in more specific the contents of each register as a function of define and use each variable. For example, in the second row, the long live variable z has been used from R_5 to define another long live variable Z_4. However, a short live variable $t_1(T_2)$ has been defined in R_8 and it will be only used on the next step by statement $t_1 \leftarrow t_1(T_1)+t_1(T_2)$. Table 5.6 and Table 5.7 present our application of FP for every short live variable which leads to a reduction in number of registers. For performing divisor doubing in *recent* coordinates, applying OSFPs results in a reduction in number of register by 1. However, for the other cases of perfroming divisor doubling/addition in projective and new weighted coordinates, OSFPs could lead to a reduction in the number of regsisters by 4.

5.5 Architecture Synthesis of an HECC Processor

This section explains the application of the architecture synthesis methods described in chapter 3 for improving the formula presented in [59, 57, 60] for genus 2 curves over fields with even characteristics. Four main approaches were followed:

1. Reduce the number of registers required. Because registers are usually used for storing intermediate variables, reducing the number of registers as much as possible is often a good idea, even if this change means slightly increased computational time, as was the case for the elliptic curve described in chapter 4. The final optimized implementation must be capable of emulating the conventional implementation of a global register file with a minimum number of read/write ports and registers. During the implementation, the interconnection of N register fields to M functional units may have a cost proportional to $N * M$. The register file allocation process assigned values into positions in single-ported register files. The execution of each three operand code instruction proceeds through three phases: reading input values from register file, executing the operations, and writing the results back into the register field. Because register files are single-ported, distinct values must be accessed concurrently for either reading or writing.

Algorithm 5.4 The modified register management of divisor doubling for *recent* coordinates for an even characteristic when $h_2 = 0$

Input: $[U_1, U_0, V_1, V_0, Z, z]$, $h = h_1x + h_0$, $f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U'_1, U'_0, V'_1, V'_0, Z', z'] = 2[U_1, U_0, V_1, V_0, Z, z]$.

```
1: $R_0 \leftarrow U_1,\ R_1 \leftarrow U_0, R_2 \leftarrow V_0, R_3 \leftarrow Z,\ R_4 \leftarrow z$

2: $R_5 \leftarrow SQR\,(R_4)$
3: $R_1 \leftarrow SQR\,(R_1)$
4: $R_6 \leftarrow SQR\,(R_0)$
5: $FP(MUL,\ ADD) \leftarrow MUL\,(R_4,\ f_3)$
6: $R_7 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_6)$
7: $R_6 \leftarrow MUL\,(R_5,\ f_0)$
8: $R_2 \leftarrow SQR\,(R_2)$
9: $R_2 \leftarrow ADD\,(R_2,\ R_6)$
10: $R_6 \leftarrow MUL\,(R_2,\ R_4)$
11: $R_5 \leftarrow MUL\,(R_1,\ R_5)$
12: $R_8 \leftarrow MUL\,(R_1,\ R_7)$
13: $R_4 \leftarrow MUL\,(R_4,\ R_8)$
14: $R_0 \leftarrow MUL\,(R_0,\ R_2)$
15: $FP(MUL,\ ADD) \leftarrow MUL\,(R_0,\ R_3)$
16: $R_3 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_4)$
17: $FP(SQR,\ MUL) \leftarrow SQR\,(h_1)$
18: $R_0 \leftarrow MUL\,(FP(SQR,\ MUL),\ R_5)$
19: $FP(MUL,\ ADD) \leftarrow MUL\,(R_2,\ R_7)$
20: $R_7 \leftarrow ADD\,(FP(MUL, ADD),\ R_0)$
21: $R_5 \leftarrow MUL\,(R_0,\ R_5)$
22: $R_0 \leftarrow MUL\,(R_0,\ R_6)$
23: $R_3 \leftarrow SQR\,(R_3)$
24: $U'_0 \leftarrow ADD\,(R_3,\ R_0)$
25: $R_3 \leftarrow SQR\,(R_6)$
26: $FP(MUL,\ SQR) \leftarrow MUL\,(R_2,\ V_1)$
27: $R_8 \leftarrow SQR\,(FP(MUL,\ SQR))$
28: $FP(MUL,\ ADD) \leftarrow MUL\,(R_3,\ f_2)$
29: $R_8 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_8)$
30: $FP(MUL,\ ADD) \leftarrow MUL\,(R_4,\ R_7)$
31: $R_4 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_8)$
32: $R_4 \leftarrow MUL\,(R_3,\ R_4)$
33: $FP(MUL,\ ADD) \leftarrow MUL\,(R_3,\ R_5)$
34: $R_4 \leftarrow MUL\,(FP(MUL,\ ADD),\ R_4)$
35: $R_0 \leftarrow SQR\,(h_1)$
36: $R_4 \leftarrow MUL\,(R_0,\ R_4)$
37: $R_2 \leftarrow MUL\,(R_2,\ R_3)$
38: $R_1 \leftarrow MUL\,(R_1,\ R_2)$
39: $FP(MUL,\ ADD) \leftarrow MUL\,(R_7,\ R_8)$
40: $R_1 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_1)$
41: $R_6 \leftarrow MUL\,(R_1,\ R_6)$
42: $R_6 \leftarrow MUL\,(R_0,\ R_6)$
43: $R_3 \leftarrow SQR\,(R_5)$
```

Table 5.4: RAVLA strategy of divisor doubling, *recent* coordinates in an even characteristic when $h_2 = 0$

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	Three Operand Code
U_1	U_0	V_1	V_0	Z	$\mathcal{U}[z]$	$\mathcal{D}[Z_4]$	–	–	–	$Z_4 \leftarrow z^2$
U_1	$\mathcal{U}[U_0]$, $\mathcal{D}[y_0]$	V_1	V_0	Z	z	Z_4	–	–	–	$y_0 \leftarrow U_0^2$
$\mathcal{U}[U_1]$	y_0	V_1	V_0	Z	z	Z_4	$D[t_1(T_1)]$	–	–	$t_1(T_1) \leftarrow U_1^2$
U_1	y_0	V_1	V_0	Z	$\mathcal{U}[z]$	Z_4	$t_1(T_1)$	$\mathcal{D}[t_1(T_2)]$	–	$t_1(T_2) \leftarrow f_3 z$
U_1	y_0	V_1	V_0	Z	z	Z_4	$\mathcal{U}[t_1(T_1)]$	$\mathcal{U}[t_1(T_2)]$, $\mathcal{D}[t_1]$	–	$t_1 \leftarrow t_1(T_1) + t_1(T_2)$
U_1	y_0	V_1	V_0	Z	z	$\mathcal{U}[Z_4]$	$\mathcal{D}[w_0(T_1)]$	t_1	–	$w_0(T_1) \leftarrow Z_4 f_0$
U_1	y_0	V_1	$\mathcal{U}[V_0]$, $\mathcal{D}[w_0(T_2)]$	Z	z	Z_4	$w_0(T_1)$	t_1	–	$w_0(T_2) \leftarrow V_0^2$
U_1	y_0	V_1	$\mathcal{U}[w_0(T_2)]$, $\mathcal{D}[w_0]$	Z	z	Z_4	$\mathcal{U}[w_0(T_1)]$	t_1	–	$w_0 \leftarrow w_0(T_1) + w_0(T_2)$
U_1	y_0	V_1	$\mathcal{U}[w_0]$	Z	$\mathcal{U}[z]$	Z_4	$\mathcal{D}[\bar{Z}]$	t_1	–	$\bar{Z} \leftarrow z w_0$
U_1	$\mathcal{U}[y_0]$	V_1	w_0	Z	z	$\mathcal{U}[Z_4]$, $\mathcal{D}[w_1]$	$\bar{Z}$	t_1	–	$w_1 \leftarrow y_0 Z_4$
U_1	$\mathcal{U}[y_0]$	V_1	w_0	Z	z	w_1	$\bar{Z}$	$\mathcal{U}[t_1]$	$\mathcal{D}[y_1(T_1)]$	$y_1(T_1) \leftarrow t_1 y_0$
U_1	y_0	V_1	w_0	Z	$\mathcal{U}[z]$, $\mathcal{D}[y_1]$	w_1	$\bar{Z}$	t_1	$\mathcal{U}[y_1(T_1)]$	$y_1 \leftarrow y_1(T_1)\, z$
$\mathcal{U}[U_1]$, $\mathcal{D}[s_0(T_1)]$	y_0	V_1	$\mathcal{U}[w_0]$	Z	y_1	w_1	$\bar{Z}$	t_1	–	$s_0(T_1) \leftarrow U_1 w_0$
$\mathcal{U}[s_0(T_1)]$	y_0	V_1	w_0	$\mathcal{U}[Z]$, $\mathcal{D}[s_0(T_2)]$	y_1	w_1	$\bar{Z}$	t_1	–	$s_0(T_2) \leftarrow s_0(T_1)\, Z$
$s_0(T_1)$	y_0	V_1	w_0	$\mathcal{U}[s_0(T_2)]$, $\mathcal{D}[s_0]$	$\mathcal{U}[y_1]$	w_1	$\bar{Z}$	t_1	–	$s_0 \leftarrow y_1 + s_0(T_2)$
$\mathcal{D}[w_2(T_1)]$	y_0	V_1	w_0	s_0	y_1	w_1	$\bar{Z}$	t_1	–	$w_2(T_1) \leftarrow h_1^2$
$\mathcal{U}[w_2(T_1)]$, $\mathcal{D}[w_2(T_2)]$	y_0	V_1	w_0	s_0	y_1	$\mathcal{U}[w_1]$	$\bar{Z}$	t_1	–	$w_2(T_2) \leftarrow w_2(T_1)\, w1$
$w_2(T_2)$	y_0	V_1	w_0	s_0	y_1	w_1	$\bar{Z}$	$\mathcal{U}[t_1]$, $\mathcal{D}[w_3(T_1)]$	–	$w_3(T_1) \leftarrow t_1 w_0$
$\mathcal{U}[w_2(T_2)]$	y_0	V_1	w_0	s_0	y_1	w_1	$\bar{Z}$	$\mathcal{U}[w_3(T_1)]$, $\mathcal{D}[w_3]$	–	$w_3 \leftarrow w_2(T_2) + w_3(T_1)$
$\mathcal{U}[w_2(T_2)]$	y_0	V_1	w_0	s_0	y_1	$\mathcal{U}[w_1]$, $\mathcal{D}[U_1']$	$\bar{Z}$	w_3	–	$U_1' \leftarrow w_2(T_2)\, w_1$
$\mathcal{U}[w_2(T_2)]$, $\mathcal{D}[w_2]$	y_0	V_1	w_0	s_0	y_1	U_1'	$\bar{Z}$	w_3	–	$w_2 \leftarrow w_2(T_2)\, \bar{Z}$

Table 5.5: RAVLA strategy of divisor doubling, *recent* coordinates in an even characteristic when $h_2 = 0$ (cont.)

R_1	R_2	R_4	R_5	R_6	R_7	R_8	R_9	Three Operand Code
y_0	V_1	$\mathcal{U}[s_0]$, $\mathcal{D}[U_0'(T_1)]$	y_1	U_1'	$\bar{Z}$	w_3	-	$U_0'(T_1) \leftarrow s_0^2$
y_0	V_1	$U_0'(T_1)$	y_1	U_1'	$\bar{Z}$	w_3	$\mathcal{D}[U_0']$	$U_0' \leftarrow U_0'(T_1) + w_2$
y_0	V_1	Z'	y_1	U_1'	$\mathcal{U}[\bar{Z}]$	w_3	U_0'	$Z' \leftarrow \bar{Z}^2$
y_0	$\mathcal{U}[V_1]$, $\mathcal{D}[V_1'(T_1)]$	Z'	y_1	U_1'	$\bar{Z}$	w_3	U_0'	$V_1'(T_1) \leftarrow V_1 w_0$
y_0	$\mathcal{U}[V_1'(T_1)]$, $\mathcal{D}[V_1'(T_2)]$	Z'	y_1	U_1'	$\bar{Z}$	w_3	U_0'	$V_1'(T_2) \leftarrow V_1'(T_1)^2$
y_0	$V_1'(T_2)$	Z'	y_1	U_1'	$\bar{Z}$	w_3	U_0'	$V_1'(T_3) \leftarrow f_2 Z'$
y_0	$\mathcal{U}[V_1'(T_2)]$, $\mathcal{D}[V_1'(T_4)$	Z'	y_1	U_1'	$\bar{Z}$	w_3	U_0'	$V_1'(T_4) \leftarrow V_1'(T_3) + V_1'(T_2)$
y_0	$V_1'(T_4)$	Z'	$\mathcal{U}[y_1]$, $\mathcal{D}[V_1'(T_5)]$	U_1'	$\bar{Z}$	w_3	U_0'	$V_1'(T_5) \leftarrow w_3 y_1$
y_0	$V_1'(T_4)$	Z'	$\mathcal{U}[V_1'(T_5)]$, $\mathcal{D}[V_1'(T_6)]$	U_1'	$\bar{Z}$	w_3	U_0'	$V_1'(T_6) \leftarrow V_1'(T_5) + V_1'(T_4)$
y_0	$V_1'(T_4)$	Z'	$\mathcal{U}[V_1'(T_6)]$, $\mathcal{D}[V_1'(T_7)]$	U_1'	$\bar{Z}$	w_3	U_0'	$V_1'(T_7) \leftarrow V_1'(T_6)\, Z'$
y_0	$V_1'(T_4)$	Z'	$V_1'(T_7)$	$\mathcal{U}[U_1']$, $\mathcal{D}[V_1'(T_8)]$	$\bar{Z}$	w_3	U_0'	$V_1'(T_8) \leftarrow w_2 U_1'$
y_0	$V_1'(T_4)$	Z'	$\mathcal{U}[V_1'(T_7)]$, $\mathcal{D}[V_1'(T_9)]$	$\mathcal{U}[V_1'(T_8)]$	$\bar{Z}$	w_3	U_0'	$V_1'(T_9) \leftarrow V_1'(T_8) + V_1'(T_7)$
y_0	$V_1'(T_{10})$	Z'	$V_1'(T_9)$	$V_1'(T_8)$	$\bar{Z}$	w_3	U_0'	$V_1'(T_{10}) \leftarrow h_1^{-1}$
y_0	$V_1'(T_{10})$	Z'	$\mathcal{U}[V_1'(T_9)]$, $\mathcal{D}[V_1']$	$V_1'(T_8)$	$\bar{Z}$	w_3	U_0'	$V_1' \leftarrow V_1'(T_{10})\, V_1'(T_9)$
y_0	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\bar{Z}$	w_3	U_0'	$V_0'(T_1) \leftarrow w_0\, Z'$
$\mathcal{U}[y_0]$, $\mathcal{D}[V_0'(T_2)]$	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\bar{Z}$	w_3	U_0'	$V_0'(T_2) \leftarrow V_0'(T_1)\, y_0$
$V_0'(T_2)$	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\bar{Z}$	$\mathcal{U}[w_3]$, $\mathcal{D}[V_0'(T_3)]$	$\mathcal{U}[U_0']$	$V_0'(T_3) \leftarrow w_3\, U_0'$
$\mathcal{U}[V_0'(T_2)]$, $\mathcal{D}[V_0'(T_4)]$	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\bar{Z}$	$\mathcal{U}[V_0'(T_3)]$	U_0'	$V_0'(T_4) \leftarrow V_0'(T_3) + V_0'(T_2)$
$\mathcal{U}[V_0'(T_4)]$	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\mathcal{U}[\bar{Z}]$, $\mathcal{D}[V_0'(T_5)]$	$V_0'(T_3)$	U_0'	$V_0(T_5) \leftarrow \bar{Z}\, V_0'(T_4)$
$V_0'(T_4)$	$\mathcal{U}[V_1'(T_{10})]$	Z'	V_1'	$V_1'(T_8)$	$\mathcal{U}[V_0'(T_5)]$, $\mathcal{D}[V_0']$	$V_0'(T_3)$	U_0'	$V_0' \leftarrow V_1'(T_{10})\, V_0'(T_5)$
$V_0'(T_4)$	$V_1'(T_{10})$	Z'	V_1'	z'	V_0'	$V_0'(T_3)$	U_0'	$z' \leftarrow {Z'}^2$

Table 5.6: OSFPs strategy of divisor doubling, *recent* coordinates in an even characteristic when $h_2 = 0$

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	FP	Three Operand Code
U_1	U_0	V_1	V_0	Z	$\mathcal{U}[z]$	$\mathcal{D}[Z_4]$	–	–	–	$Z_4 \leftarrow z^2$
U_1	$\mathcal{U}[U_0]$, $\mathcal{D}[y_0]$	V_1	V_0	Z	z	Z_4	–	–	–	$y_0 \leftarrow U_0^2$
$\mathcal{U}[U_1]$	y_0	V_1	V_0	Z	z	Z_4	$D[t_1(T_1)]$	–	–	$t_1(T_1) \leftarrow U_1^2$
U_1	y_0	V_1	V_0	Z	$\mathcal{U}[z]$	Z_4	$t_1(T_1)$	–	$\mathcal{D}[t_1(T_2)]$	$t_1(T_2) \leftarrow f_3 z$
U_1	y_0	V_1	V_0	Z	z	Z_4	$\mathcal{U}[t_1(T_1)]$	$\mathcal{D}[t_1]$	$\mathcal{U}[t_1(T_2)]$	$t_1 \leftarrow t_1(T_1) + t_1(T_2)$
U_1	y_0	V_1	V_0	Z	z	$\mathcal{U}[Z_4]$	$\mathcal{D}[w_0(T_1)]$	t_1	–	$w_0(T_1) \leftarrow Z_4 f_0$
U_1	y_0	V_1	$\mathcal{U}[V_0]$, $\mathcal{D}[w_0(T_2)]$	Z	z	Z_4	$w_0(T_1)$	t_1	–	$w_0(T_2) \leftarrow V_0^2$
U_1	y_0	V_1	$\mathcal{U}[w_0(T_2)]$, $\mathcal{D}[w_0]$	Z	z	Z_4	$\mathcal{U}[w_0(T_1)]$	t_1	–	$w_0 \leftarrow w_0(T_1) + w_0(T_2)$
U_1	y_0	V_1	$\mathcal{U}[w_0]$	Z	$\mathcal{U}[z]$	Z_4	$\mathcal{D}[\bar{Z}]$	t_1	–	$\bar{Z} \leftarrow z w_0$
U_1	$\mathcal{U}[y_0]$	V_1	w_0	Z	z	$\mathcal{U}[Z_4]$, $\mathcal{D}[w_1]$	$\bar{Z}$	t_1	–	$w_1 \leftarrow y_0 Z_4$
U_1	$\mathcal{U}[y_0]$	V_1	w_0	Z	z	w_1	$\bar{Z}$	$\mathcal{U}[t_1]$	$\mathcal{D}[y_1(T_1)]$	$y_1(T_1) \leftarrow t_1 y_0$
U_1	y_0	V_1	w_0	Z	$\mathcal{U}[z]$, $\mathcal{D}[y_1]$	w_1	$\bar{Z}$	t_1	$\mathcal{U}[y_1(T_1)]$	$y_1 \leftarrow y_1(T_1)\, z$
$\mathcal{U}[U_1]$, $\mathcal{D}[s_0(T_1)]$	y_0	V_1	$\mathcal{U}[w_0]$	Z	y_1	w_1	$\bar{Z}$	t_1	–	$s_0(T_1) \leftarrow U_1 w_0$
$\mathcal{U}[s_0(T_1)]$	y_0	V_1	w_0	$\mathcal{U}[Z]$	y_1	w_1	$\bar{Z}$	t_1	$\mathcal{D}[s_0(T_2)]$	$s_0(T_2) \leftarrow s_0(T_1)\, Z$
$s_0(T_1)$	y_0	V_1	w_0	$\mathcal{D}[s_0]$	$\mathcal{U}[y_1]$	w_1	$\bar{Z}$	t_1	$\mathcal{U}[s_0(T_2)]$	$s_0 \leftarrow y_1 + s_0(T_2)$
$s_0(T_1)$	y_0	V_1	w_0	s_0	y_1	w_1	$\bar{Z}$	t_1	$\mathcal{D}[w_2(T_1)]$	$w_2(T_1) \leftarrow h_1^2$
$\mathcal{D}[w_2(T_2)]$	y_0	V_1	w_0	s_0	y_1	$\mathcal{U}[w_1]$	$\bar{Z}$	t_1	$\mathcal{U}[w_2(T_1)]$	$w_2(T_2) \leftarrow w_2(T_1)\, w1$
$w_2(T_2)$	y_0	V_1	w_0	s_0	y_1	w_1	$\bar{Z}$	$\mathcal{U}[t_1]$	$\mathcal{D}[w_3(T_1)]$	$w_3(T_1) \leftarrow t_1 w_0$
$\mathcal{U}[w_2(T_2)]$	y_0	V_1	w_0	s_0	y_1	w_1	$\bar{Z}$	$\mathcal{D}[w_3]$	$\mathcal{U}[w_3(T_1)]$	$w_3 \leftarrow w_2(T_2) + w_3(T_1)$
$\mathcal{U}[w_2(T_2)]$	y_0	V_1	w_0	s_0	y_1	$\mathcal{U}[w_1]$, $\mathcal{D}[U_1']$	$\bar{Z}$	w_3	–	$U_1' \leftarrow w_2(T_2)\, w_1$
$\mathcal{U}[w_2(T_2)]$, $\mathcal{D}[w_2]$	y_0	V_1	w_0	s_0	y_1	U_1'	$\bar{Z}$	w_3	–	$w_2 \leftarrow w_2(T_2)\, \bar{Z}$

Table 5.7: OSFPs strategy of divisor doubling, *recent* coordinates in an even characteristic when $h_2 = 0$ (cont.)

R_1	R_2	R_4	R_5	R_6	R_7	R_8	FP	Three Operand Code
y_0	V_1	$\mathcal{U}[s_0]$, $\mathcal{D}[U_0'(T_1)]$	y_1	U_1'	$\bar{Z}$	w_3	-	$U_0'(T_1) \leftarrow s_0^2$
y_0	V_1	$U_0'(T_1)$	y_1	U_1'	$\bar{Z}$	w_3	–	$U_0' \leftarrow U_0'(T_1) + w_2$
y_0	V_1	Z'	y_1	U_1'	$\mathcal{U}[\bar{Z}]$	w_3	–	$Z' \leftarrow \bar{Z}^2$
y_0	$\mathcal{U}[V_1]$	Z'	y_1	U_1'	$\bar{Z}$	w_3	$\mathcal{D}[V_1'(T_1)]$	$V_1'(T_1) \leftarrow V_1 w_0$
y_0	$\mathcal{D}[V_1'(T_2)]$	Z'	y_1	U_1'	$\bar{Z}$	w_3	$\mathcal{U}[V_1'(T_1)]$	$V_1'(T_2) \leftarrow V_1'(T_1)^2$
y_0	$V_1'(T_2)$	Z'	y_1	U_1'	$\bar{Z}$	w_3	$\mathcal{D}[V_1'(T_3)]$	$V_1'(T_3) \leftarrow f_2 Z'$
y_0	$\mathcal{U}[V_1'(T_2)]$, $\mathcal{D}[V_1'(T_4)$	Z'	y_1	U_1'	$\bar{Z}$	w_3	–	$V_1'(T_4) \leftarrow V_1'(T_3) + V_1'(T_2)$
y_0	$V_1'(T_4)$	Z'	$\mathcal{U}[y_1]$	U_1'	$\bar{Z}$	w_3	$\mathcal{D}[V_1'(T_5)]$	$V_1'(T_5) \leftarrow w_3 y_1$
y_0	$V_1'(T_4)$	Z'	$\mathcal{D}[V_1'(T_6)]$	U_1'	$\bar{Z}$	w_3	$\mathcal{U}[V_1'(T_5)]$	$V_1'(T_6) \leftarrow V_1'(T_5) + V_1'(T_4)$
y_0	$V_1'(T_4)$	Z'	$\mathcal{U}[V_1'(T_6)]$, $\mathcal{D}[V_1'(T_7)]$	U_1'	$\bar{Z}$	w_3	–	$V_1'(T_7) \leftarrow V_1'(T_6)\, Z'$
y_0	$V_1'(T_4)$	Z'	$V_1'(T_7)$	$\mathcal{U}[U_1']$	$\bar{Z}$	w_3	$\mathcal{D}[V_1'(T_8)]$	$V_1'(T_8) \leftarrow w_2 U_1'$
y_0	$V_1'(T_4)$	Z'	$\mathcal{U}[V_1'(T_7)]$, $\mathcal{D}[V_1'(T_9)]$	U_1'	$\bar{Z}$	w_3	$\mathcal{U}[V_1'(T_8)]$	$V_1'(T_9) \leftarrow V_1'(T_8) + V_1'(T_7)$
y_0	$V_1'(T_{10})$	Z'	$V_1'(T_9)$	$V_1'(T_8)$	$\bar{Z}$	w_3	–	$V_1'(T_{10}) \leftarrow h_1^{-1}$
y_0	$V_1'(T_{10})$	Z'	$\mathcal{U}[V_1'(T_9)]$, $\mathcal{D}[V_1']$	$V_1'(T_8)$	$\bar{Z}$	w_3	–	$V_1' \leftarrow V_1'(T_{10})\, V_1'(T_9)$
y_0	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\bar{Z}$	w_3	–	$V_0'(T_1) \leftarrow w_0\, Z'$
$\mathcal{U}[y_0]$, $\mathcal{D}[V_0'(T_2)]$	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\bar{Z}$	w_3	–	$V_0'(T_2) \leftarrow V_0'(T_1)\, y_0$
$V_0'(T_2)$	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\bar{Z}$	$\mathcal{U}[w_3]$	$\mathcal{D}[V_0'(T_3)]$	$V_0'(T_3) \leftarrow w_3\, U_0'$
$\mathcal{U}[V_0'(T_2)]$, $\mathcal{D}[V_0'(T_4)]$	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\bar{Z}$	w_3	$\mathcal{U}[V_0'(T_3)]$	$V_0'(T_4) \leftarrow V_0'(T_3) + V_0'(T_2)$
$\mathcal{U}[V_0'(T_4)]$	$V_1'(T_{10})$	Z'	V_1'	$V_1'(T_8)$	$\mathcal{U}[\bar{Z}]$, $\mathcal{D}[V_0'(T_5)]$	$V_0'(T_3)$	–	$V_0(T_5) \leftarrow \bar{Z}\, V_0'(T_4)$
$V_0'(T_4)$	$\mathcal{U}[V_1'(T_{10})]$	Z'	V_1'	$V_1'(T_8)$	$\mathcal{U}[V_0'(T_5)]$, $\mathcal{D}[V_0']$	$V_0'(T_3)$	–	$V_0' \leftarrow V_1'(T_{10})\, V_0'(T_5)$
$V_0'(T_4)$	$V_1'(T_{10})$	Z'	V_1'	z'	V_0'	$V_0'(T_3)$	–	$z' \leftarrow {Z'}^2$

2. Reduce the number of read/write operations for the register file. If the number of read/write operations from the register file can be reduced, the consumption of processing power will also decrease. Most explicit formulas accomplish this goal by reducing the number of finite field operations. However, with the methods presented in this thesis, further improvement was obtained through the selection of faster explicit formulas, special register allocation via variable liveness analysis, the combination of multiplications and squaring or multiplications and addition operations using an operation scheduling via forwarding paths, efficient register spilling through storage binding, and the overlapping of multiplication and addition or square operations whenever there is no datapath dependence.

3. Reduce the amount of computation time. The goal here was to use a special operation scheduling algorithm that makes use of the operation of the squaring and addition units, which consumes only one clock cycle, in parallel with finite field multiplication which consumes at least half of the field size. Although the performance of the design is not affected, this approach reduces the total number of clock cycles needed for computing one divisor addition or one divisor doubling. Further improvement was also obtained through an analyzing the liveness of the control unit and through overlapping the divisor addition and divisor doubling operations within one divisor multiplication, which leads to a reduction in the computational time either in the divisor addition and divisor doubling level or in the divisor multiplication level.

4. Reduce the amount of storage memory. The goal here was to apply the new storage binding algorithm, which rearranges the storing of the intermediate variables throughout the divisor addition and divisor doubling operations, keeping the results of either the divisor doubling or divisor addition operations in registers if they are to be used later in the next explicit formula. This storage occurs continually because the intermediate results between the doubling and the additions operations are usually computed towards the end of the explicit formula. Keeping some of the results in registers that are be used by the next loop of the divisor multiplication operation thus reduces the amount of storage memory that is being used to store the intermediate variables between divisor operations.

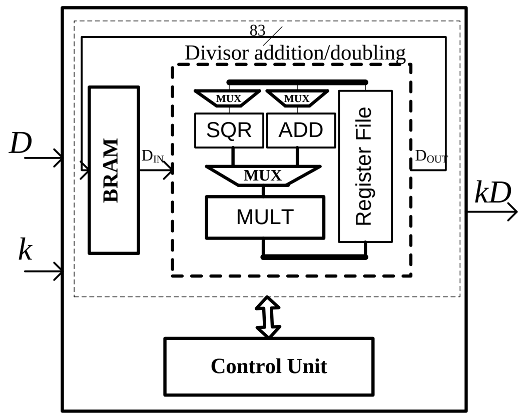


Figure 5.1: Basic architecture of Hyperelliptic Curve Divisor Implementation

5.5.1 Conventional HECC processor

The HECC processor is, in fact, the hardware architecture for divisor multiplication. The basic architecture consists of a control unit, block RAM (BRAM), and a divisor addition/doubling unit, as shown in Figure 5.1. The architecture of divisor addition/doubling unit is comprised of four classes of blocks: finite field multiplier, finite field arithmetic (i.e. squaring and adder), register file for intermediate results storage, and multiplexers for selection and forwarding intermediate results. The control unit receives the HECC parameters, reads a key (or a scalar), and controls the divisor addition/doubling unit according to the binary double and add divisor multiplication algorithm shown in section 2.3.5. The divisor addition/doubling unit, on the other hand, is responsible for computing all required field arithmetic operations. At the beginning of the scalar multiplication operation, the BRAM is assumed to contain the scalar and a divisors. These values must be maintained during the iterations of the HECC scalar multiplication.

For HECP, the processor must compute the scalar multiplication kD. **Efficient algorithms are required in order to implement the group operations as well as for the finite field arithmetic. The work for this thesis included the development of different prototypes of processor architectures that are well suited for implementation in field programmable gate arrays (FPGAs). For example, the user can choose to have two finite field adders, one finite field multiplier, one finite field squarer, etc. The conventional HECC processor consists of four main components: the control unit (CU), the field arithmetic unit (FAU), the register file**

(RF), and the multiplexers (MUXs). The Control unit generates control signals for the RF, FAU, and MUX. The MUXs govern the data exchange between the RF and the FAU. The FAU performs the field and group operations as described in chapter 2. The following subsections provide a more detailed description of the other individual components.

5.5.1.1 Register File

A register file (RF) is one of the main blocks of digital electronic systems used in computers, communications, and embedded systems. Flip-flops are a fundamental building block of an RF. In synchronous systems, they are responsible for storing and sampling data with respect to the clock signal for sequential circuit designs in both pipeline systems and state machines. In asynchronous systems, flip-flops often act as synchronizes that interface between two unrelated signals that may operate at different frequencies and phases. Traditional RF designs have focused primary on a balanced design tradeoff between delay and power, as indicated by the optimum power-delay-product (PDP) value. In this work, register liveness analysis was incorporated into the HECC processor design along with performance and power consumption considerations in order to produce a high-performance and low-power design. Register liveness analysis is an optimization technique that has been used to minimize the number of read/write ports of RF and has been described in chapter 3.

Multi-Port Register File This register file has two write ports, including one asynchronous write port, and two read ports, including one asynchronous read port. Involving such a register file accomplishes multiple goals for tuning the area, power, and performance of register files per execution iteration of the algorithm. The optimized specification for the HECC processor is expressed as a three-operand code for a conventional implementation with a multi-port register file. In optimized implementations, positions in a conventional HECC processor register file are referred to as variables. A three-operand code operation $R_1 \Leftarrow R_2 + R_3$ is said to define variable stores in register R_1 and to use variables store in registers R_2 and R_3. Using a multi-port register file for storing the values of the variables creates problems. One problem is associated with computing the minimum number of ports required to access as many variables as needed. If each variable always accesses the register file through the same port, then the problem is reduced. Storage binding, which was described in section 3.2.3 has been used to binding variables to ports. In this work, a register file is assumed to have two ports for either read or write, thus requiring one cycle per access.

5.5.1.2 Multiplexers

The multiplexer (MUX), one of the basic data path connection elements, contributes a significant amount of area costs especially for FPGA designs. A recent study from Altera, based on the analysis of 100 customer designs, stated that multiplexer account for 26% of the logic element utilization. Optimizing multiplexer use is therefore very important for the overall quality of digital designs. Behavioural synthesis, which compiles designs specified in high-level languages into register-transfer level code, determines the main micro-architecture of designs and thus has a significant impact on the quality of a design. Behavioural synthesis consists of three basic stages: register allocation, operation scheduling, and storage binding. The allocation determines how many instances of each type of resource (functional units or registers) are needed; scheduling determines when a computational operation is executed; storage binding assigns operations (or variables) to either the memory or the register. Each of these steps has an influence on multiplexer utilization. In this work, the first two steps were assumed to be already completed, and the focus was on only the third step, storage binding, in order to optimize multiplexer.

Internal Data forwarding A very serious difficulty with a genus 2 explicit formula is the number of registers used to store the intermediate variables involved. An implementation of the divisor addition operation that does not incorporate consideration of register requirements would use up to 30 registers. The size of each register is the same as of the field size. This number of registers is too large for constrained environments. The area of the register file can be further reduced through the use of internal data forwarding among multiple functional units. An internal data forwarding conflict is a situation which an algorithm statement refers to the data if a preceding statement. The technique used to discover the internal data forwarding conflicts among statements is called dependance analysis. Consider two algorithm statements S_1 and S_2 with S_1 occurring before S_2. The three possible types of conflicts are true dependance, load dependency, and anti dependency.

A true dependency, also known as read after write, occurs when an statement depends on the result of a previous statement. In other words, statement S_1 tries to read a source variable before statement S_2 writes it, so a statement S_2 incorrectly gets the old value. A load dependency, also known as write after write, occurs when the ordering of statements affects the final load value of a variable. For example, statement S_2 tries to write an operand before it is written by statement S_1. The write operation is being performed in the wrong order, leaving the value in the destination written by statement S_1 rather than by S_2. An anti dependency, also known as write before read, occurs when a statement read a variable that is later updated by a write operation. Statement S_2 tries to write destination value before it is read by statement S_1, so statement S_2 incorrectly gets the new value. This conflict causes some statements write results early in the algorithm and other statements read operands late in the algorithm. For optimal dependance analysis, data forwarding conflicts must be traced through choosing an optimal statement scheduling.

5.6 Architecture Optimization for an HECC Processor

To demonstrate the accuracy of the newly derived formulas, all of the group operations introduced in this thesis were implemented on an Xilinx FPGA. An additional goal was to show that an HECC can achieve the same performance level as an ECC and, in some cases, even outperform an ECC. A variety of hardware architectures for genus 2 HECC were investigated, including ones that use projective, new weighted, and recent coordinate systems that can consider a group order of approximately 2^{80}. The optimization of the HECC was investigated at three levels: the field operation level, the group operation level, and the scalar multiplication level. The study included analysis of tradeoffs between pluralization options, latency, read/write power consumption, and area/time optimized configurations.

5.6.1 Macroscopic Structural View of HECC Datapath

A datapath is an interconnection of resources, which implement finite field arithmetic or logic functions; steering logic circuits, e.g., multiplexers and buses,

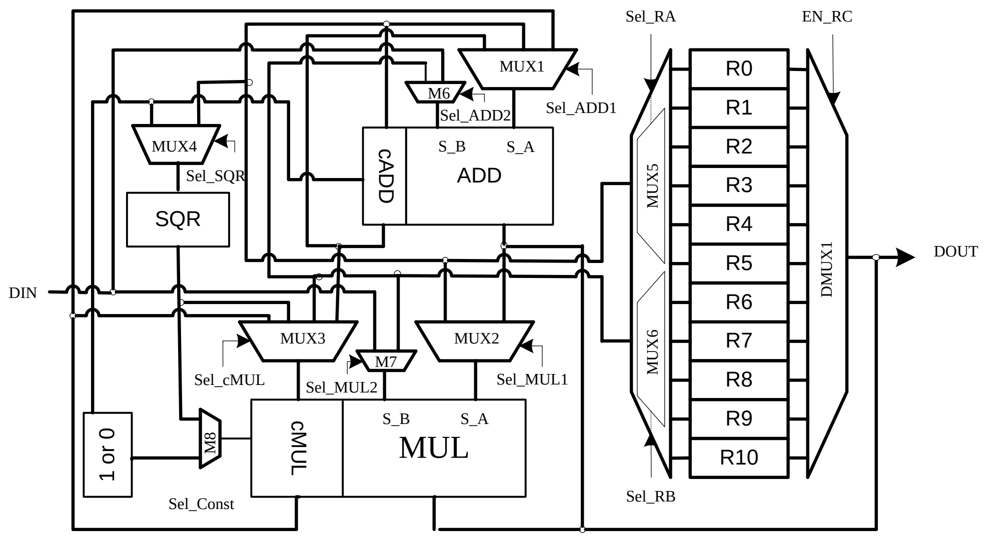


Figure 5.2: Structure view of new weighted coordinate divisor doubling datapath

which send data to the appropriate destination at the appropriate time; and registers or memory arrays for storing data , input/output ports and the control unit. For example, the inputs, outputs and controls of the multiplexers have to be specified and connected. An example of one possible macroscopic structure for an HECC processor is shown in Figure 5.2 and 5.3.

Figure 5.2 shows a refined view of the datapath of the new weighted coordinate divisor doubling with 11 registers, one multiplier, one squarer and one adder. It shows explicit the interconnection of the multiplexers. For simplicity, registers have been used to store constants 1 or 0. Obviously, these registers can be changed into a hard-wired implementation. The blocks cADD and cMUL have been used to perform an addition and a multiplication operation with constant, respectively. The connections to the input/output ports are not shown. Addition multiplexers are required to load registers with input data from memory. The connections to the control unit are the enable signals for all registers, the selectors of the multiplexers. The multiplier returns signal done to the control unit, detecting the completion of one finite field multiplication.

5.7 Hyperelliptic Curve Datapath Analysis

This section presents an FPGA implementation that includes consideration of explicit formula based on *projective*, *new weighted*, and *recent* coordinates. The

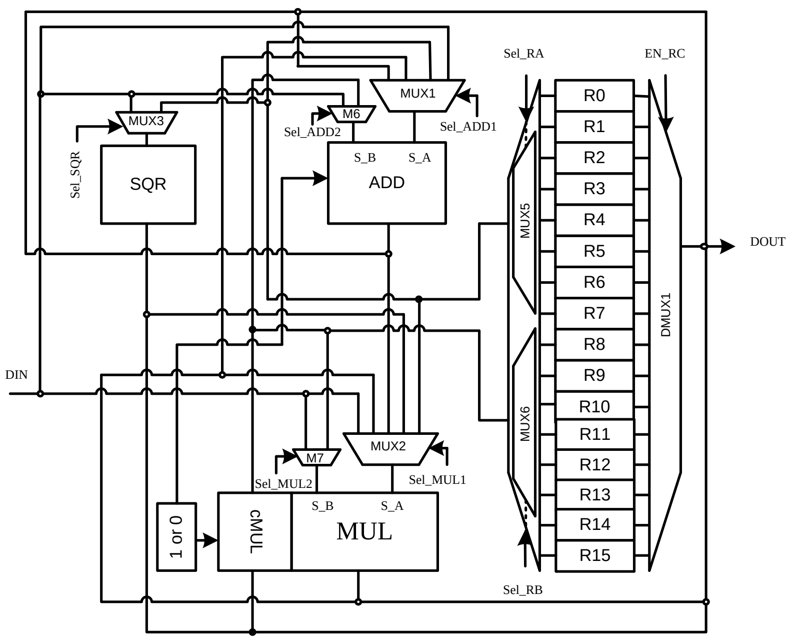


Figure 5.3: Structural view of new weighted coordinates divisor mixed addition datapath

goal of this section is to describe general datapath analysis for HECC processor targeted for hardware implementation. Datapath analysis has been used to study the area, power and performance of the processor architectures and their implementation in FPGA logic.

5.7.1 Area of the HECC processor

Table 5.8 specifying the area complexities, the critical path delay, and the power consumed by each datapath component in HECC processor architecture. The performance of a circuit is specified in terms of the circuit critical path delay. A circuit critical path delay corresponds to the longest combinatorial delay of the circuit. The critical path delay of a circuit defines the maximum clock frequency F_{max} at which it can operate. When more resources are available, it is possible to lower the critical path. With the use of digit-serial multiplier, as described in section 2.1, the finite field multiplication can be performed twice as fast as the bit-serial multiplier. The main finding is that, for area optimization architecture, the design based on a bit-serial multiplier is preferable compared to those using digit-serial multiplier.

In this work have synthesized the proposed HECP using FPGA. We have writ-

Table 5.8: Area, power consumption and clock rate for Data path component

Data Path	Area				Clock Rate		Power (mW)	
Component	#FF	#LUT	Slices	#IOs	CLK ns	F_{max} MHz	Leakage	Dynamic
Bit Multiplier_83	258	352	182	253	3.489 (2)	287.45	1350.93	84.43
Digit Multiplier_83 (G = 2)	450	540	292	212	2.288 (1)	437.10	1355.14	131.73
Digit Multiplier_83 (G = 4)	812	924	512	192	2.288 (1)	437.10	1354.36	156.02
Digit Multiplier_83 (G = 8)	1496	1688	944	182	2.996 (2)	333.78	1352.00	181.08
Digit Multiplier_83 (G = 16)	2912	3216	1808	177	2.971 (2)	336.61	1350.41	267.56
Field Squarer_83	-	84	48	168	6.170 (3)	162.07	1343.51	0.37
MUX2t1_83	-	83	60	250	3.207 (3)	311.81	1343.57	0.99
MUX3t1_83	-	83	83	334	6.532 (4)	153.09	1344.18	7.96
MUX4t1_83	-	166	83	417	6.814 (4)	146.75	1344.28	9.07
Register File_83 (R = 13)	1092	1464	960	441	9.845 (7)	101.57	1355.70	139.65
Register File_83 (R = 15)	1260	1592	1152	443	9.977 (7)	100.23	1355.74	138.85
Register File_83 (R = 16)	1344	1656	1248	446	10.043 (8)	99.57	1355.77	138.46
Register File_83 (R = 17)	1428	1720	1344	447	10.109 (8)	98.92	1355.82	138.07

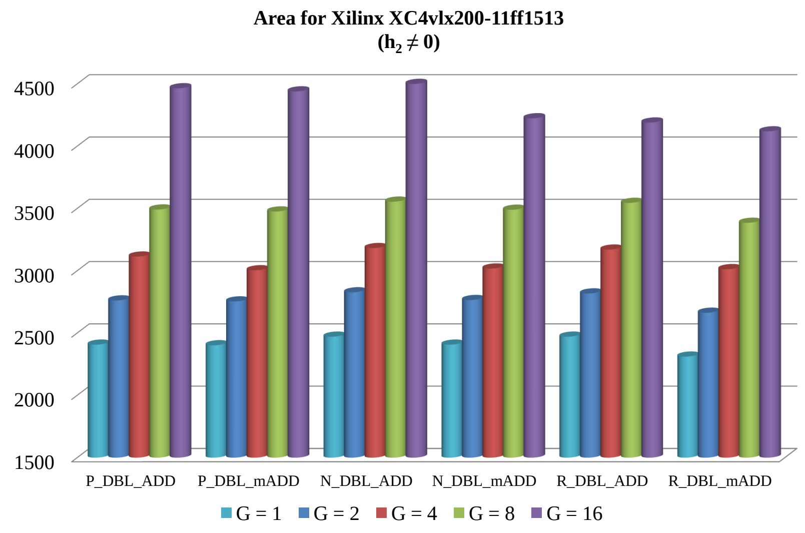


Figure 5.4: Area in number of slices for different digit-serial multiplier when $h_2 \neq 0$

ten in VHDL the RTL description and synthesized it using Xilinx XC4vlx200-11ff1513. Figure 5.4 and Figure 5.5 shows the required area for the proposed HECP for different inversion free coordinates when $h_2 \neq 0$ and $h_2 = 0$, respectively. As in the figure, the required areas are proportional to the digit size G of the finite field multiplier.

5.7.2 Energy Consumption of the HECC Processor

The relationship between energy and power is that power is defined as energy used over time, i.e. the rate at which energy is expanded. In other words, power is energy divided by time. Power is measured in milliWatts, while energy is measured in micro joules. The following simple equation governs dynamic power consumption:

$$\textbf{Dynamic Power} = CV^2f$$

where C is the capacitance of the node switching, V is the supply voltage, and f is the switching frequency. If capacitance and supply voltage are not changing, then more nodes switching at higher frequencies, the dynamic power would tend

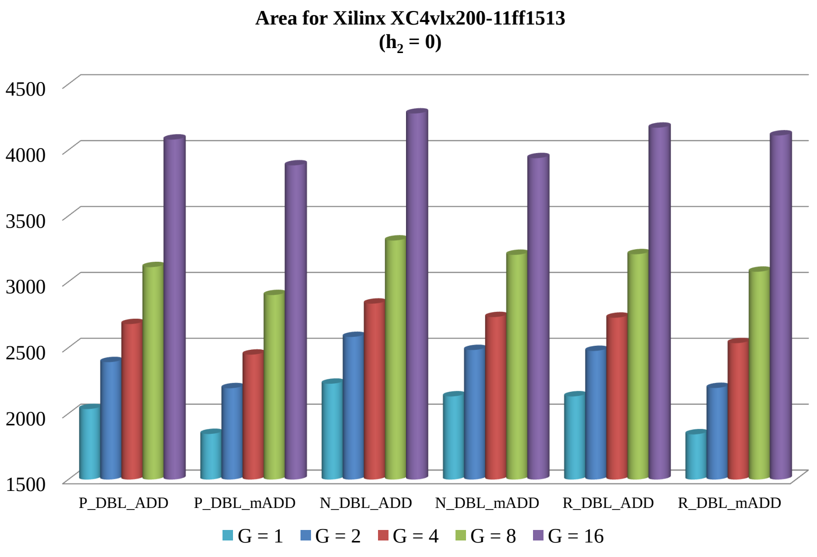


Figure 5.5: Area in number of slices for different digit-serial multiplier when $h_2 = 0$

to increase.

The design of low-energy cryptographic hardware is an active research area due to the demand for portable applications of cryptographic algorithms. Power reduction techniques have been proposed for all levels of the design hierarchy, from algorithmic and architectural optimizations to circuit and technological innovations. Reference [98] proposed that switching activities dominate and contribute to more than 90% of the total power consumption. Reducing switching activities has therefore become the major target in attempts to power reduction. Pipelining has been used to reduce both the critical path computation time and switching activities. Rather than using pipelining, this work demonstrates that, as shown in Figure 5.6 and Figure 5.7, power reduction can achieved by reformulating the explicit formulas for performing HECC algorithms through the modification to the register allocation, operation scheduling, and storage binding procedures. Power is also reduced by increasing the parallelism of the finite field multiplication operation and by rearranging the datapath topology through a reduction in the number of read/write operations from/to the register file using forwarding paths between finite field operations.

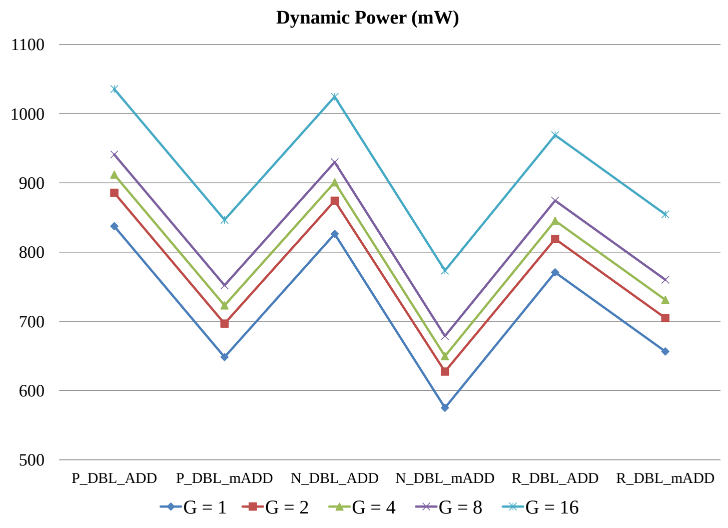


Figure 5.6: Power consumption in (mW) for different digit-serial multiplier when $h_2 \neq 0$

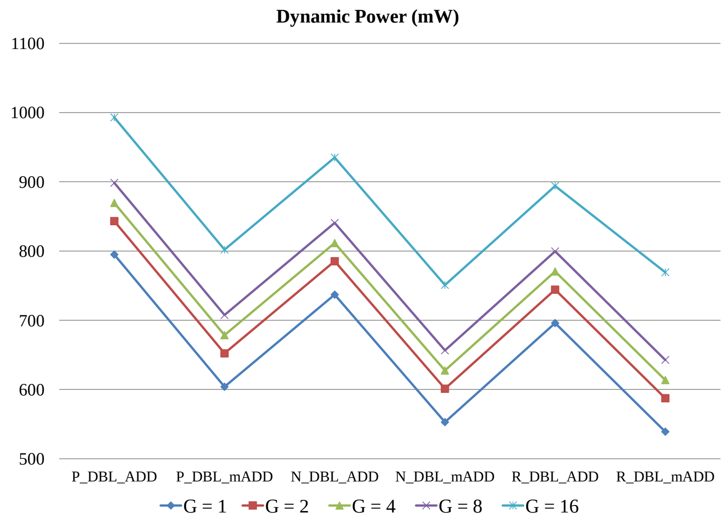


Figure 5.7: Power consumption in (mW) for different digit-serial multiplier when $h_2 = 0$

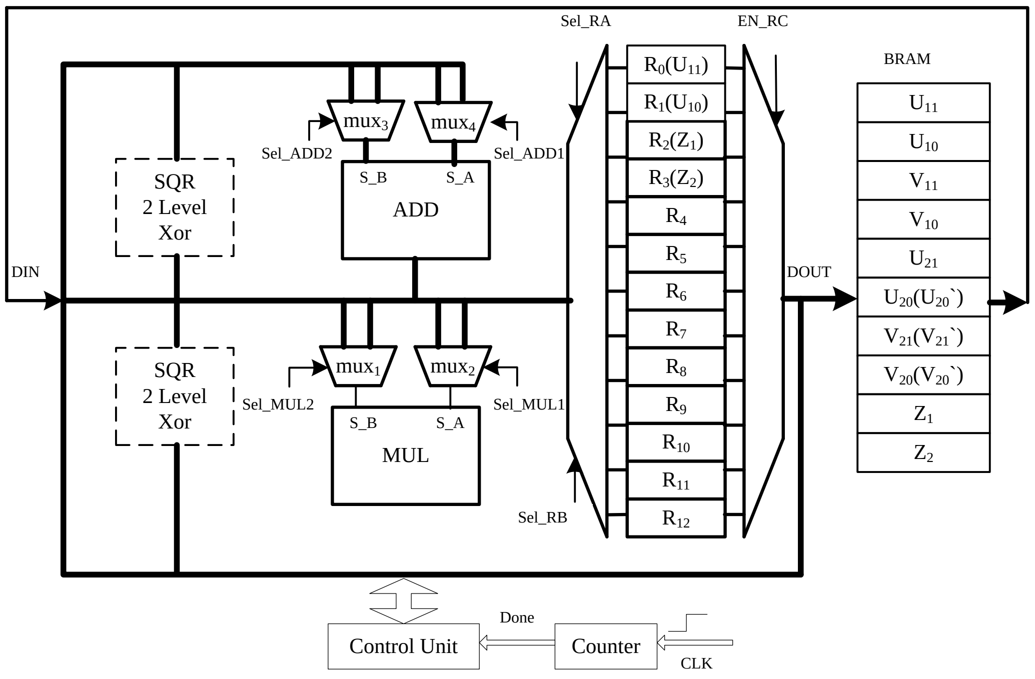


Figure 5.8: Top-level Data path for Projective Coordinates Divisor Doubling-Addition

5.7.3 Improve the performance of the HECC processor

In this section we describe our work on improving the performance of computing divisor multiplication for the HECC processor. The idea of reducing the critical path was the starting point for our improvement. Critical path plays an important role in the performance of a design on FPGAs. Good timing performance requires that the critical path delay be as efficient as possible. It is often acceptable to let the area increase within a tolerance limit if the timing could improved. This approach involved slightly increase the number of registers and reducing the size of the multiplexers. Even though, for the most part, reducing area and power result against this, changes can be made in order to improve the performance. Synthesis results concluded that it was indeed faster to use more registers.

5.8 Hyperelliptic Curve Control Analysis

The control unit liveness of shared resources is the main focus of this section. The new approach to liveness analysis was achieved through two steps: first,

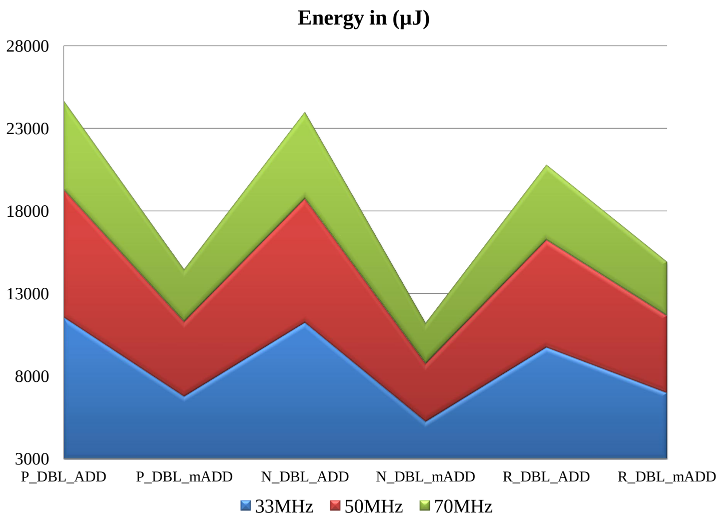


Figure 5.9: Energy in μJ for different maximum frequency when $h_2 \neq 0$

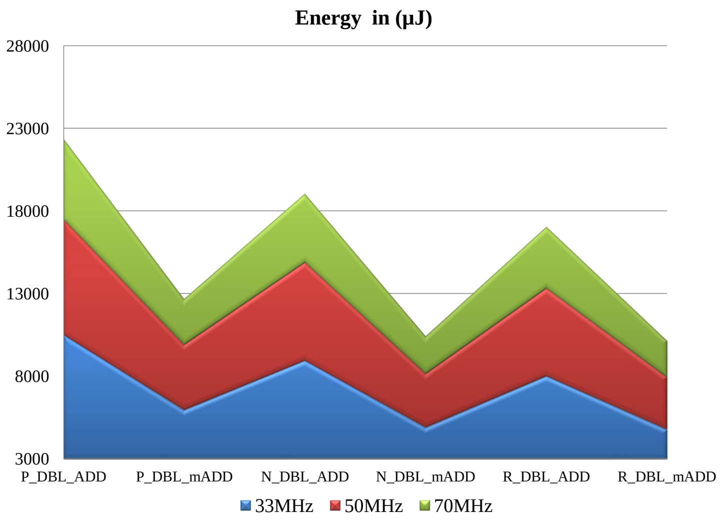


Figure 5.10: Energy in μJ for different maximum frequency when $h_2 = 0$

by graphing the busy states of the shared resources, and second, by using this graph to drive the formal analysis of the divisor multiplication specification. If the goal of a designer is to lower the number of registers required, he can trade the number of registers for additional latency; however, he should avoid overlapping and should start a group operation only when the previous one has finished. It should be noted that a single group operation uses from 8 to 14 registers and that the maximum number of registers is reached when two group operations overlap.

To achieve the primary goal of reducing the number of registers, lower register usage was chosen over for additional latency. In most cases, meeting this goal means that the design should avoid overlapping between datapaths and should start a divisor doubling operation, as shown in Figure 5.2, only when the previous divisor doubling or mixed addition, as shown in Figure 5.3, has been completed. This work demonstrates that overlapping a group operation, as shown in Figure 5.11, is in fact necessary for minimizing the number of registers and can be achieved after a specific liveness analysis of the resource controllers under the divisor multiplication level, which previously described in section 3.3.2. For example, in the case of projective coordinates, the output consists of five coefficients: five field elements that are neither produced at the same time nor required for the initiation of the next group operation. This feature means that once one field element has been computed, it can then be used by the next group operation.

A number of optimizations can be used to improve the proposed design which previously described in section 5.6. Namely, we draw the control flow diagram as shown in Figure 5.11, so that it achieved overlapping between group operation as well as field operation. The advantage of having overlapping was that less computation time needed to be controlled.

5.9 Experimental Results

This section presents the results obtained from the implementation of HECC on an FPGA and ASIC. FPGAs have potential advantages over ASIC implementations in cryptographic applications for measuring dynamic power as well as algorithm agility, speed facility, architecture efficiency, resource efficiency, ability to modify the algorithm, efficient throughput, and cost efficiency. Some effort has

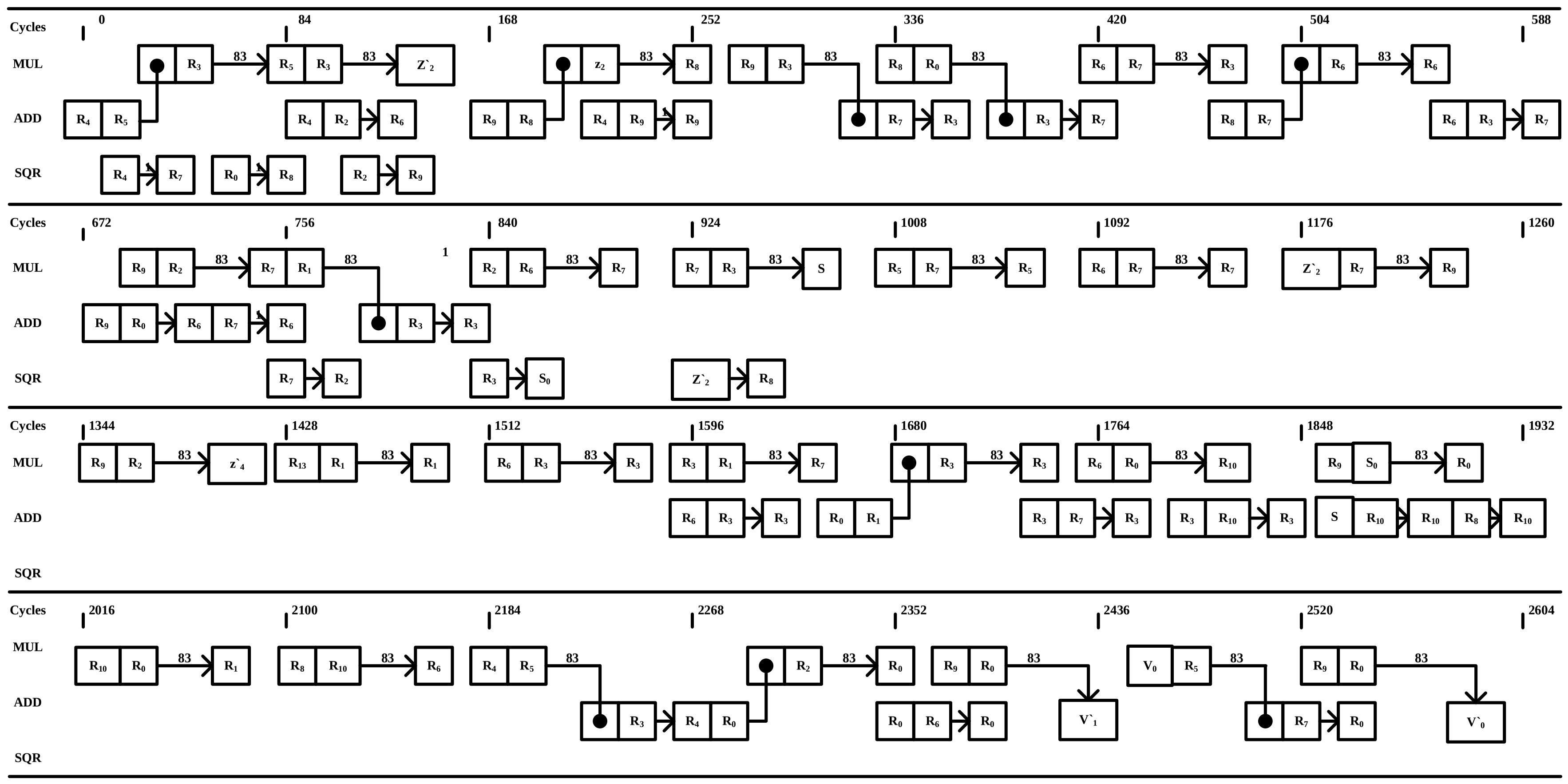


Figure 5.11: An example of a divisor doubling operation with overlapping

therefore been done to implement HECC on hardware devices, such as FPGAs. Most of the implementations, however, are based on the algorithm introduced by Cantor and not on an the explicit formula. Accelerating this curve-based arithmetic is possible in several ways, especially multiplication, choosing a coordinate representation that is more efficient and increasing the speed of a divisor multiplication operation.

In this work, Lange's explicit formulas of even characteristic and a special polynomial $h(x)$ where $h_2 \neq 0$ and $h_2 = 0$, were used to develop an efficient explicit formula for HECC. The focus was on techniques for optimizing an implementation to make it suitable for constrained devices, mainly through a reduction in area and power consumption. Register liveness analysis, combined with data forwarding paths, was used in order to reduce the number registers from 23 to 14 for HECC new coordinate divisor addition, from 19 to 14 for mixed HECC new coordinate divisor addition, and from 16 to 14 in HECC new coordinate divisor doubling. As a result, the overall HECC processor requires from 40,021 to 49,819 gate element and the total divisor multiplication time is limited between 10.08 ms to 15.82 ms.

The original goal of designing the HECP was to reduce the overall area. In order to accomplish this, pipelining was excluded from the design since adding piped stages would significantly increase the area or the resultant design. The strategy pursued involved composing architecture synthesis processes for the different inversion-free explicit formula with the goal of reducing the number of registers. After obtaining the register management, architecture optimizations would be made, based on analysis the tradeoff between area, power, and computation time.

An interesting new technology that has evolved in the early 2000s is that of creating an ASIC directly from an FPGA-based design. Many designers use FPGAs for ASIC prototyping [90]. They use automated tools to implement their circuit on FPGAs, and they then extensively test the circuit in the circuit's environment, for example, in a prototype digital video player or a prototype satellite communication chip. The FPGA-based prototype may be larger, costlier, and more power-hungry than an ASIC-based implementation, but can be useful for detecting and correcting errors in the circuit, for creating other components and software that interact with the circuit. Once satisfied with the circuit, automated tools could be used to reimplemented the circuit on an ASIC. The ASIC

implementation traditionally did not utilize any information from the FPGA implementation.

Table 5.10 and Table 5.9 show the results obtained with optimized explicit formulas for a number of different inversion-free coordinate systems after synthesis. The first column indicates the coordinate design system, and the remainder of the table is divided into two sections. The center set of columns shows the results using Xilinx Vertix-4 FPGAs, and the right-hand set shows the results using synthesis tools based on ASIC compiler design. The first column shows the coordinate design system.

In the FPGA section, the first column shows the numbers of flip-flops used for synthesizing each design, while the second column shows the number of LUTs. The next two columns present the number of slices and the computation time in terms of number of cycles.

Table 5.9: FPGA and ASIC Synthesis and Simulate Power Efficiency Results for HECC Divisor Multiplication

Design	FPGA				ASIC		
	Leakage (mW)	Dynamic (mW)	#Cycles	FPGA$_{Energy}$	Time (ms)	Power (mW)	ASIC$_{Energy}$
Projective Coordinates ($\mathcal{P}$)							
$\mathcal{P}_{DBL-ADD_{h_2\neq 0}}$	794.96	164.23	857,391	15,174.38	15.82	713.77	11,291.84
$\mathcal{P}_{DBL-ADD_{h_2=0}}$	784.52	146.93	809,816	14,046.26	15.08	677.64	10,218.81
$\mathcal{P}_{DBL-mADD_{h_2\neq 0}}$	786.00	163.04	632,930	11,426.44	12.04	526.62	6,340.50
$\mathcal{P}_{DBL-mADD_{h_2=0}}$	765.19	144.29	589,347	10,277.12	11.30	490.49	5,542.53
New Weighted Coordinates ($\mathcal{N}$)							
$\mathcal{N}_{DBL-ADD_{h_2\neq 0}}$	800.25	165.32	841,121	15,043.58	15.58	701.73	10,932.95
$\mathcal{N}_{DBL-ADD_{h_2=0}}$	794.96	148.42	736,596	13,094.11	13.88	617.29	8,567.98
$\mathcal{N}_{DBL-mADD_{h_2\neq 0}}$	786.05	163.05	548,825	10,041.48	10.58	454.08	4,804.16
$\mathcal{N}_{DBL-mADD_{h_2=0}}$	780.78	146.50	523,285	9,476.80	10.22	435.95	4,455.40
Recent Coordinates ($\mathcal{R}$)							
$\mathcal{R}_{DBL-ADD_{h_2\neq 0}}$	790.25	159.42	773,210	13,561.28	14.28	647.90	9,252.01
$\mathcal{R}_{DBL-ADD_{h_2=0}}$	789.57	141.86	691,312	12,238.99	13.14	581.60	7,642.22
$\mathcal{R}_{DBL-mADD_{h_2\neq 0}}$	775.56	155.58	639,754	11,397.15	12.24	537.58	6,579.97
$\mathcal{R}_{DBL-mADD_{h_2=0}}$	765.19	138.58	510,628	9110.00	10.08	429.28	4,327.14

Table 5.10: FPGA and ASIC Synthesis and Simulate Area Efficiency Results for HECC Divisor Multiplication

Design	FPGA				ASIC		
	#FF	#LUT	#Slices	Time(ms)	#Cells	#Gates	Area$_{mm^2}$
Projective Coordinates ($\mathcal{P}$)							
$\mathcal{P}_{DBL-ADD_{h_2\neq0}}$	2267	3456	2407	17.16	27,614	48,040	66.328
$\mathcal{P}_{DBL-ADD_{h_2=0}}$	1850	3079	2035	16.44	21,719	42,799	46.477
$\mathcal{P}_{DBL-mADD_{h_2\neq0}}$	2267	3584	2402	14.26	26,615	49,819	66.296
$\mathcal{P}_{DBL-mADD_{h_2=0}}$	1682	2951	1846	13.54	19,747	41,020	40.501
New Weighted Coordinates ($\mathcal{N}$)							
$\mathcal{N}_{DBL-ADD_{h_2\neq0}}$	2351	3520	2471	16.92	27,601	48,930	67.525
$\mathcal{N}_{DBL-ADD_{h_2=0}}$	2018	3207	2227	15.22	23,691	44,579	52.806
$\mathcal{N}_{DBL-mADD_{h_2\neq0}}$	2267	3424	2407	13.30	25,616	47,595	60.956
$\mathcal{N}_{DBL-mADD_{h_2=0}}$	1934	3143	2131	12.8	22,705	43,689	49.598
Recent Coordinates ($\mathcal{R}$)							
$\mathcal{R}_{DBL-ADD_{h_2\neq0}}$	3772	4881	3595	15.84	28,725	48,466	66.339
$\mathcal{R}_{DBL-ADD_{h_2=0}}$	1934	3143	2131	14.50	22,705	43,689	49.598
$\mathcal{R}_{DBL-mADD_{h_2\neq0}}$	2099	3328	2311	12.38	24,642	46,261	49.309
$\mathcal{R}_{DBL-mADD_{h_2=0}}$	1682	2951	1843	14.50	19,746	40,021	39.512

Chapter 6

Comparison of Elliptic and Hyperelliptic

Elliptic curve cryptosystems (ECCs) are widely used in parctical applications. Cryptgraphic protocols based on ECC are fast and cosidered as a secure alternative to other common PKCs like RSA [78]. Hyperelliptic curve cryptosystems (HECCs) have had very little practical relevance, in part because the group operations on HECC are belived to have a relatively high computational cost. Recent developments of shorter explicit formula and high performance implementations could turn the trend from ECC towards HECC [96]. In this chapter, the emphasis lies on the comparison of the complexity of ECP and HECP based on FPGA hardware implementation. A comparsion will be derived to state whether a HECP or an ECP is efficient in terms of energy and time/area factors. For simplicity we restrict ourselves to characteristic two and a security of 80 bits. Therefore, we deal with field sizes of 163 bits for ECC, 83 for HECC of genera two. We also will be able to compare the expected register requirements, area, performance, and energy factors of different cryptosystems depending on the bit sizes and the properties of the implemented recommended NIST field libraries. The presented comparsion can be carried over to arbitrary characteristic and arbitrary field sizes.

Table 6.1: Complexity of ECC and HECC processors for the 80 bit level of security

Characteristic	$\mathcal{LD}_{DBL-ADD} - \mathcal{P}_{DBL-ADD_{h_2 \neq 0}}$			$\mathcal{LD}_{DBL-mADD} - \mathcal{P}_{DBL-mADD_{h_2 \neq 0}}$		
	ECC	HECC	$\frac{\text{ECC}}{\text{HECC}}$	ECC	HECC	$\frac{\text{ECC}}{\text{HECC}}$
# FF	3327	2267	1.47	2652	2267	1.17
# LUT	4917	3456	1.42	3856	3584	1.08
# Slices	3120	2407	1.29	2582	2402	1.07
Power (mW)	468.57	713.77	0.65	234.24	526.62	0.44
Time (ms)	6.14	17.16	0.35	4.71	14.26	0.33
Energy (μJ)	2877.02	12248.29	0.23	1103.27	7509.60	0.15

6.1 Complexity of Elliptic and Hyperelliptic Curve Processors

Developing hardware implementation of cryptographic algorithms for ultra-low area and power devices is not as straightforward as compiling existing VHDL code for an low area and power FPGA or ASIC library. We carefully selected several algorithms that seemed promising for an ultra-low area and power implementation. We made extensive use of area and power saving techniques on the architectural, logic, and system level when we implemented the algorithms. In most cases the speed of the algorithm is not as important as the area and energy. At the low clock frequencies of these devices leakage power is dominant. Therefore, we minimized the energy by minimizing the circuit size.

A fair comparision between ECC and HECC is difficult to achieve due to the different field sizes, types of operations, and the non-deterministic nature of the HECC operations, in particular, the computation of polynomial Greatest Common divisors when using Cantor's algorithm. In addition, a lot of the published ECC results contain many platform specific optimizations which vary greatly between different implementations. The metric we use for measuring area, power and computation time on ECC and HECC over characteristic two fields is based on a ratio of time/area and energy characteristics tradeoffs rather than bit complexity or specific timings. Depending on efficient hardware implementation through the application of proposed arcitecture synthesis and optimization methodology, we will be able to determine in which cases ECP is time/area and energy efficient than HECP in terms of inversion-free coordinates and vice versa.

Table 6.1 compares the two inversion-free coordinate $\mathcal{LD}_{DBL-ADD}$ and $\mathcal{P}_{DBL-ADD}$ when $h_2 = 0$ or $h_2 \neq 0$. A few observations can be made from the table. First, the energy consumption and the computation time are lower in ECC for the $\mathcal{LD}_{DBL-ADD}$ than those of HECC for $\mathcal{P}_{DBL-ADD}$. Another observation is that the combined #FF and #LUT for $\mathcal{P}_{DBL-ADD}$ significantly decreases the area requirements compared to the results obtained for $\mathcal{LD}_{DBL-ADD}$. In addtion we see that under certain conditions genus 2 hyperelliptic curves can be more area efficient than ECC at the same level of security. This result, however, implies to use very specific curves for HECC.

Our efforts to verify the correctness of the above mentioned implementations included a comparison of theoretical and practical results based upon bottom-up implementation for both ECP and HECP. Finite field, point or divisor arithmetic components are coded and verified first. These components are then interconnected using structural style descriptions to form the top-level component of the scalar multiplication is implemented. This approach comprises a bottom-up implementation. Each component should be defined so that it is individually verifiable. Each component, either finite field arithmetic and/or divisor or point arithmetic, is independently verified before being combined with other components to form a scalar multiplication component. Scalar multiplication component is independently verified before being combined to form the complete system. The complete system is then verified.

6.2 Elliptic Curve NIST-recommended Comparisons

In previous chapters, we discussed the methods for reducing the number of registers for different inversion-free coordinates when performing scalar multiplication either based on ECC or HECC. Another case where register file would be a problem is the size of each register. Because it is impossible not to perform any field operations on a partition field size element, as well as provide as less as possible for the whole field element to become stable on the field arithmetic operation, it is important that the register size should be equal to the field size. While increasing field size requires extra in register size to store the intermediate results, a fixed register size algorithm is requires but the computation time may be larger. If we carefully design an ECC processor with fixed register size, decreasing rate of the area can be larger than that of increasing rate of computation time when field size is increased. For the following discussion, we will use the fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$, and $\mathbb{F}_{2^{571}}$ for ECC. We are focus on a comparison of the register requirments, area, power, and computational complexity.

In February 2000, FIPS 186-1 was revised by NIST to include the elliptic curve digital signature algorithm (ECDSA) as specified in ANSI X9.62 [1] with further recommendations for the selection of underlying finite fields and elliptic curves; the revised standard is called FIPS 186-2 [2]. For binary fields $\mathbb{F}_{2^m}$, m was chosen so that there exists a random curve of almost prime order over $\mathbb{F}_{2^m}$. FIPS 186-2 has 5 recommended finite fields over binary fields: $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$, and $\mathbb{F}_{2^{571}}$. For each of the binary fields one randomly selected elliptic curve

Table 6.2: NIST recommended binary finite fields and their reduction polynomials

Binary finite field	Reduction polynomial
$\mathbb{F}_{2^{163}}$	$F(x) = x^{163} + x^7 + x^6 + x^3 + 1$
$\mathbb{F}_{2^{233}}$	$F(x) = x^{233} + x^{74} + 1$
$\mathbb{F}_{2^{283}}$	$F(x) = x^{283} + x^{12} + x^7 + x^5 + 1$
$\mathbb{F}_{2^{409}}$	$F(x) = x^{409} + x^{87} + 1$
$\mathbb{F}_{2^{571}}$	$F(x) = x^{571} + x^{10} + x^5 + x^2 + 1$

and one Koblitz curve was selected, see for example [37]. Table 6.2 lists the NIST recommended five binary fields and their reduction polynomials.

Based on the above discussion, this work proposes a fixed register file size that can be implemented by only w bits independent of m bits of the field size $\mathbb{F}_{2^m}$ so that registers for storing intermediate computational variables have only w bits. Moreover, the work presented in this chapter takes different field sizes into account and analyzes the result by comparing it with area, power and timings measured on hardware. In addition this work explains that using fixed register file size can yield small ECP area compared to conventional ECP with variable register filed size based on the field size. For the theoretical comparison we use López-Dahab projective and mixed coordinates on curves over different finite fields. To our knowledge López-Dahab coordinates are usually the system of choice for hardware implementation of elliptic curves over binary fields.

In this work, for a field $\mathbb{F}_{2^m}$ and a register size of $w = 83\,\text{bit}$ López-Dahab projective coordinate requires $\lceil \frac{m}{83} \rceil \times 4$ number of registers needed to store intermediate variables of operations for a scalar multiplication depending on the underlying field size m. However, López-Dahab mixed coordinates requires $\lceil \frac{m}{83} \rceil \times 2$ number of registers. The disadvantage is the increased number of computation cycles which incresed the total computation time of performing the scalar multiplication. Table 6.3 shows the results are 6.149 ms to 71.390 ms for a scalar multiplication using López-Dahab projective coordinates on curves over different finite fields. Based on our number of cycles calculation in Table 2.2, it has been increased by 27,038 and 20,647 cycles for $\mathcal{LD}_{projective}$ and $\mathcal{LD}_{mixed}$, respectively.

Table 6.3 presents the total energy consumption and execution time values of the $\mathcal{LD}_{DBL-ADD}$ and $\mathcal{LD}_{DBL-mADD}$ for varying field sizes. A lower number of registers is used to store the intermediate variables. A few important observations can be made from the table. First, the energy consumption as well as the computation time values monotonically increase with the increase in the field size.

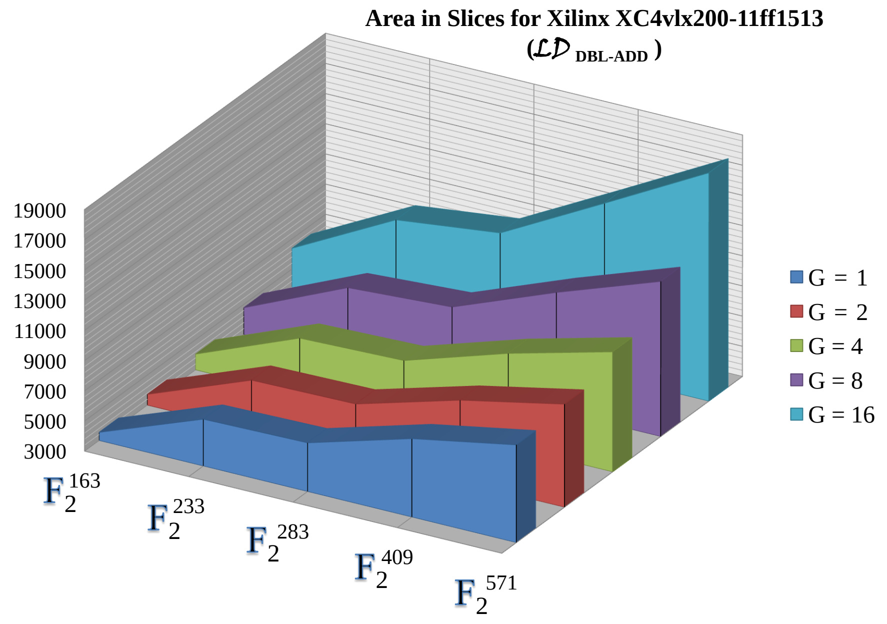


Figure 6.1: Area in number of slices for different digit-serial multipliers for $\mathcal{LD}_{DBL-ADD}$

This is because the larger the field size, the more register objects are allocated and the more the clock cycles are needed.

On a closer look, it is observed that the energy and the execution values increase in a stepwise manner. Second, it is observed that for $\mathcal{LD}_{DBL-mADD}$ the energy values are lower than the corresponding energy values for $\mathcal{LD}_{DBL-ADD}$. This implies that the reduction of the number of registers has more impact on reducing the total energy consumption. Lastly, it is observed that for each field size the normalized execution time values are lower than the corresponding energy consumption values. This implies that the reduction of the number of registers has more impact on reducing the total energy consumtion of the design than on reducing the computation time. This observation is justified because the difference in energy per access to the register is larger than the diferences in its access times.

Figure 6.1 and Figure 6.2 depict the area in slices values for all NIST recommended field sizes for $\mathcal{LD}_{DBL-ADD}$ and $\mathcal{LD}_{DBL-mADD}$, respectively. From Figure 6.1, a couple of observations can be made for $\mathcal{LD}_{DBL-ADD}$ using different digit-serial sizes for the finite field multiplier. First, increases in the digit-serial and the field sizes both cause an increase in the area in slices. Second, the comparison of the $\mathcal{LD}_{DBL-ADD}$ with the $\mathcal{LD}_{DBL-mADD}$, in Figure 6.2, reveals that for the digit-size $G = 16$, the area in slices are reduced by about 15%.

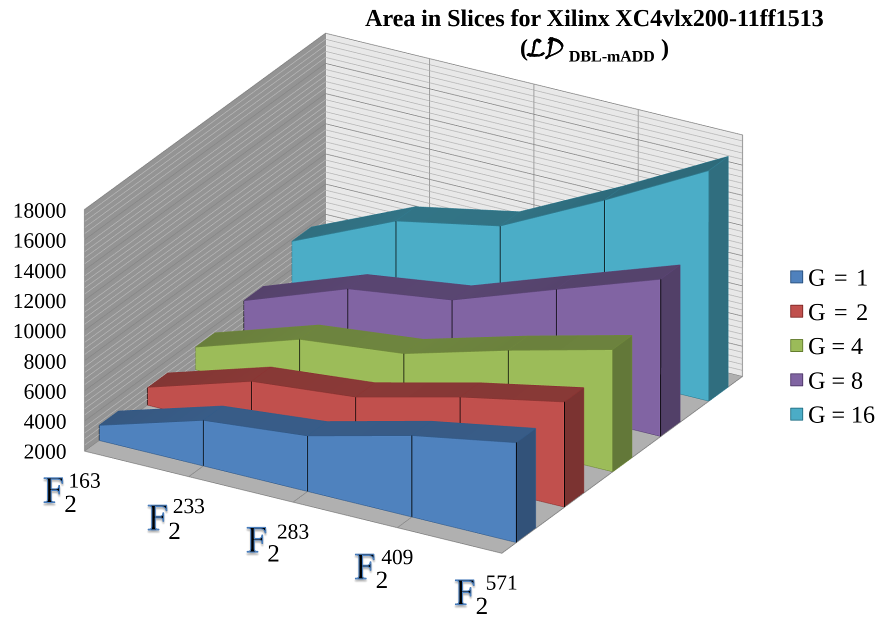


Figure 6.2: Area in number of slices for different digit-serial multipliers for $\mathcal{LD}_{DBL-mADD}$

Table 6.3: Estimated area, power, and computation time comparison for other NIST-recommended finite field

Finite Field	#Reg	#FF	#LUT	#Slices	#Cycles	Time(ms)	Power(mW)	Energy(μJ)
López-Dahab Projective Coordinates ($\mathcal{LD}_{DBL-ADD}$)								
$\mathbb{F}_{2^{163}}$	8	3327	4917	3120	307,473	6.149	468.57	2881.45
$\mathbb{F}_{2^{233}}$	12	6110	8866	5504	614,339	12.286	723.91	8894.52
$\mathbb{F}_{2^{283}}$	16	6721	9540	5436	899,400	17.988	945.75	17,012.15
$\mathbb{F}_{2^{409}}$	20	9290	11612	6938	1,847,637	36.952	1223.94	45,227.93
$\mathbb{F}_{2^{571}}$	28	10273	14455	7878	3,569,538	71.390	1793.82	128,062.17
López-Dahab Mixed Projective Coordinates ($\mathcal{LD}_{DBL-mADD}$)								
$\mathbb{F}_{2^{163}}$	4	2652	3856	2582	235,148	4.702	234.24	1101.62
$\mathbb{F}_{2^{233}}$	6	4902	7470	4425	469,844	9.396	370.95	3485.77
$\mathbb{F}_{2^{283}}$	8	5382	8479	4894	687,820	13.756	472.85	6504.71
$\mathbb{F}_{2^{409}}$	10	6430	10442	6141	1,412,012	28.240	611.97	17,282.17
$\mathbb{F}_{2^{571}}$	14	9558	12734	7023	2,727,268	54.545	884.91	48,267.73

Chapter 7

Conclusion and Future Direction

Security is a critical factor for these ultra-low power devices due to their impact on privacy, trust and control. Wireless sensor network (WSN) belongs to a new set of ultra-low power applications which make computing ubiquitous. WSN is quickly becoming a vital part of our infrastructure [23].These applications impose severe power and area constraints on the underlying hardware devices. Traditional cryptographic algorithms are considered too bulky, complex and power hungry for these devices. The goal our research was to develop an efficient hardware implementation for these constrained devices.

In this thesis we have considered architecture, algorithms and arithmetic for curve-based cryptography, more precisely ECC and HECC. Hyperelliptic curves, a generalization of elliptic curves, require smaller field sizes as genus increases. Hyperelliptic curves of genus g achieved equivalent security of ECC with field size $\frac{1}{g}$ times the size of field of ECC for $g \leq 2$. For example, a genus 2 HECC with an operand length of 83 bits provides the same level of security than ECC with an operand length of 163 bits. Over the past few years, a number of researchers have attempted to increase the efficiency of curve-based cryptography arithmetic. The group operations are considerably more complex. The most time consuming operation in ECC and in HECC is the ECSM and HCDM respectively. The emphasis of this work has been on efficient hardware implementations with awareness of architecture synthesis and optimization and on applications with resource constrained devices. There exist several ways to accelerate the curve-based arithmetic such as: speeding up the finite field arithmetic, choosing a suitable coordinates system for efficient group operation and accelerating the ECSM or HEDM operations. Most of these aspects are considered in previous works, but mainly aimed towards efficient software implementations. Nevertheless, a hardware processor is a necessity in some constrained application for minimize the area, to lower the power consumption, and/or to speed-up the processing time.

First, we have focused on modeling architecture synthesis factors that should lead to a reduced register requirements. Starting from providing an efficient explicit formulas for performing group operations in different inversion-free coordinate systems. Second, we have developed the macroscopic structure view of both ECP and HECP datapath for the different coordinates systems. Third, we have generated the VHDL source code to synthesis and simulate using FPGA and ASIC technologies. Fourth, we have discussed the possibilities of improving the design through datapath and control unit analysis and investigate the effectiveness on area, power and computation time. Finally, we compared between HECP and ECP for the same level of security in terms of area, computation time, and energy.

It is widely recognized that data security will play a central role in future information technology systems. The use of elliptic curve in cryptography can be found in many applications. Elliptic curve algorithms are used for services like key establishment and digital signatures. Hyperelliptic curve based cryptosystems have been enjoying increasing attention in recent years. They have long been considered as not competitive with elliptic curve based counter parts because of construction, but the situation has changed in the last few years. It is now possible to efficiently construct hyper elliptic curve whose Jacobian has cryptographically good order. Long operands are a major drawback in embedded applications such as cellular phones, etc., where memory and processing power are constrained. For curves of genus two over binary fields, they require only 80 to 120-bit arithmetic compared to numbers that are typically 160-256 bits in length for elliptic curve cryptosystems, which seem promising for many applications.

When constrained devices are to be used for implementing a public key cryptographic system, one can use a smaller key size ECP implementation. We have shown here a small area implementation of an ECP in both projective and mixed coordinates. Using register liveness analysis, we have observed approximately 38% area savings at the point arithmetic level. The ECP has been implemented using FPGAs. An improved register management scheme has been developed for point multiplication using the L$\acute{o}$pez-Dahab projective and mixed coordinates. We have also used forwarding paths that lead to a reduction in area and dynamic power. A point multiplication computational time of $1652.88\,\mu s$ and $1271.99\,\mu s$ have been achieved for projective and mixed coordinates respectively. These results show that optimizing the number of write backs into the register file, analyzing the data dependency of intermediate variables and scheduling the scalar multiplication operations must be carefully handled to save area and power.

7.1 Future Direction

Hardware implementations of cryptographic algorithms are vulnerable to side-channel attacks. Side-channel attacks that are based on multiple measurements of the same operation can be potentially countered by employing masking techniques. Many protection measurements depart from an idealized hardware model that is very expensive to meet with real hardware. In what follows, we discuss some of the related open problems that can provide a basis for future work.

- This work has focused on the hardware implementation of the divisor arithmetic for genus ≤ 2. However, it is important to address the implementation of genus > 2 including protection against side-channel attacks.
- We have presented a modified register management of differenet explicit formula for genus $\leq$ 2 hyperelliptic curves in projective, new weighted and recent coordinates. These formula allow inversion-free group arithmetic for those curves. Such group operations are particularly interesting for hardware implementations since we remove the need for an inverter unit, which can produce significant savings in the area of the processor, or allow to have more multiplier units for high performance. Developing a modfied register management for genus > 2 can be a subject of future work.
- In this work, we have reduced the minimum number of registers required to store the intermediate variables for both genus 1 and 2. However, the significant increase in the number of intermediate variables for genus 3 and 4 can be an interesting future work for proposing an efficient register alloca-

tion, operation scheduling and storage binding aiming an efficient number of register requirements.

- **In this work, we have proposed a special model for register allocation, operation scheduling which can be practically implemented in software for defining an efficient explicit formulas for hyperelliptic curves of genus higher than 2.**
- **In this work, we have developed a stand alone hyperelliptic curve processor for performing divisor multiplication based on inversion-free coordinates. However, formal verification, which means a mathematical proof of the correctness of a certain design for all possible inputs to result a correct output, can be an interesting future work for both elliptic and hyperelliptic curve processors based on a special architecture synthesis models.**

Bibliography

[1] ANSI X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), 1999.

[2] National Institute of Standards and Technology, Digital Signature Standard, FIPS publication 186-2, February 2000.

[3] Implementing cryptographic pairings on smartcards. In *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop*, pages 134–147, 2006.

[4] IEEE Std 1363-2000. IEEE standard specifications for public-key cryptography. IEEE Computer Society, August 2000.

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

[6] Bijan Ansari and Anwar Hasan. High-Performance Architecture of Elliptic Curve Scalar Multiplication. *IEEE Transactions on Computers*, 57(11):1443–1453, 2008.

[7] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C.* 2004.

[8] R. Avanzi, N. Thériault, and Z. Wang. Rethinking low genus hyperelliptic jacobian arithmetic over binary fields: Interplay of field arithmetic and explicit formulae. Technical report, Centre for Applied Cryptographic Research (CACR), available at http://www.cacr.math.uwaterloo.ca/techreports/2006/cacr2006-07.pdf., CACR 2006-07.

[9] M. Aydos, T. Yanik, and C.K. Koc. High-speed implementation of an ECC-based wireless authentication protocol on an ARM microprocessor. *Communications, IEE Proceedings-*, 148(5):273–279, 2001.

[10] L. Batina, J. Guajardo, T. Kerins, N. Mentens, P. Tuyls, and I. Verbauwhede. An elliptic curve processor suitable for RFID-tags. *Cryptology ePrint Archive*, Report 2006/227, 2006.

[11] Lejla Batina, Nele Mentens, Kazuo Sakiyama, Bart Preneel, and Ingrid Verbauwhede. Public-key cryptography on the top of a needle. *In Proc. IEEE International Symposium on Circuits and Systems (ISCAS07), Special Session on Novel Cryptographic Architectures for Low-Cost RFID*, pages 1831–1834, 2007.

[12] G. Bertoni, L. Breveglieri, and M. Venturi. Power aware design of an elliptic curve coprocessor for 8 bit platforms. In *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on*, pages 5–341, 2006.

[13] Guido Bertoni, Jorge Guajardo, Sandeep Kumar, Gerardo Orlando, Christof Paar, Thomas Wollinger, and Gerardo Orlando. Efficient GF(p^m) arithmetic architectures for cryptographic applications. In *TOPICS IN CRYPTOLOGY - CT RSA 2003*, pages 158–175. Springer-Verlag, 2003.

[14] I. F Blake, G. Seroussi, and N. P Smart. Elliptic curves in cryptography. *London Mathematical Society Lecture Note Series, 265, Cambridge University Press*, 1999.

[15] N. Boston, T. Clancy, Y. Liow, and J. Webster. Genus two hyperelliptic curve coprocessor. In *In Workshop on Cryptographic Hardware and Embedded Systems / CHES 2002*, pages 400–414. Springer-Verlag, 2002.

[16] David G. Cantor. Computing in the Jacobian of a hyperelliptic curve. *Mathematics of Computation*, 48(177):95–101, 1987.

[17] A.P. Chandrakasan and R. W. Brodersen. *Low Power Digital CMOS Design.* Kluwer Academic Publishers, 1995.

[18] William N. Chelton and Mohammed Benaissa. Fast Elliptic Curve Cryptography on FPGA. *IEEE transactions on Very Large Scale Integrated (VLSI) systems*, 16(2):198–205, February 2008.

[19] Hyun Min Choi, Chun Pyo Hong, and Chang Hoon Kim. FPGA implemntation of high performance elliptic curve cryptographic processor over GF(2^{163}). *Journal of Systems Architecture*, 54(10):893–900, April 2008.

[20] T. Clancy. Analysis of FPGA-based hyperelliptic curve cryosystems. Master's thesis, University of Illinois, Urbana-Champaign, Illinois, 2002.

[21] T. Clancy. FPGA-based hyperelliptic curve cryptosystems. Invited paper presented at AMS Central Section Meeting, 2003.

[22] Henri Cohen and Gerhard Frey, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, volume 34 of *Discrete Mathematics and Its Applications*. Chapman & Hall/CRC, 2005.

[23] David E. Culler and Hans Mulder. Smart sensors to network the world. In *Scientific American*, pages 84-91, June 2004.

[24] Jan Denef and Frederik Vercauteren. An extension of Kedlaya's algorithm to hyperelliptic curves in characteristic 2, 2002.

[25] Jean-Pierre Deschamps. *Hardware Implementation of Finite-Field Arithmetic*. McGraw-Hill Professional, 1st edition, February 2009.

[26] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[27] Sylvain Duquesne and Oberthur Card Systems. Classification of genus 2 curves over $\mathbb{F}_{2^n}$ and optimization of their arithmetic. cryptology eprint archive: Report 2004/107.

[28] Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A survey of Lightweight-Cryptography implementations. *IEEE Des. Test*, 24(6):522–533, 2007.

[29] Grace Elias, Ali Miri, and Tet-Hin Yeap. On efficient implementation of FPGA-based hyperelliptic curve cryptosystems. *Comput. Electr. Eng.*, 33:349–366, September 2007.

[30] M. Fayed, M.W. El-Khamshi, and F. Gebali. A high-speed, low-area processor array architecture for multiplication over GF(2^m). In *International Conference on Microelectronics (ICM)*, pages 61–64, Cairo, December 2007.

[31] G. Gaubatz, J. Kaps, and B. Sunar. Public key cryptography in sensor networks - revisited. *In 1st European Workshop on Security in Ad-Hoc and Sensor Networks (ESAS 2004)*, August 2004.

[32] G. Gaubatz, J.-P. Kaps, E. Ozturk, and B. Sunar. State of the art in ultra-low power public key cryptography for wireless sensor networks. In *Pervasive Computing and Communications Workshops, 2005. PerCom 2005 Workshops. Third IEEE International Conference on*, pages 146–150, 2005.

[33] Pierrick Gaudry and Robert Harley. Counting points on hyperelliptic curves over finite fields. In *Proceedings of the 4th International Symposium on Algorithmic Number Theory*, ANTS-IV, pages 313–332, London, UK, UK, 2000. Springer-Verlag.

[34] J. Goodman and A.P. Chandrakasan. An Energy-efficient Reconfigurable Public-key Cryptography Processor. *IEEE Journal of Solid-State Circuits,*, 36(11):1808–1820, 2001.

[35] James Ross Goodman. Energy scalable reconfigurable cryptographic hardware for portable applications. 2000. AAI0802715.

[36] Jorge Guajardo, Tim GÃneysu, Sandeep Kumar, Christof Paar, and Jan Pelzl. Efficient hardware implementation of finite fields with applications to cryptography. *Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications*, 93(1):75–118, 2006.

[37] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, New York, USA, 2004.

[38] R. Harley. Fast arithmetic on genus two curves. Technical report, Available at http://cristal.inria.fr/harley/hyper, adding.txt and doubling.c, 2000.

[39] Anwar Hasan. *ECE 720: Selected Topics in Cryptographic Computations Lecture Notes.* University of Waterloo, 2008.

[40] J. Heyszl and F. Stumpf. Efficient One-pass Entity Authentication Based on ECC for Constrained Devices. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 88–93, Anaheim, CA, July 2010.

[41] J. Chao K. Matsuo and S. Tsujii. Fast genus two hyperelliptic curve cryptosystems. Technical report, ISEC2001-23, IEICE, pages 89-96, 2001.

[42] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Sov. Phys. Dokl. (English translation), vol. 7, no. 7, pp. 595-596*, 1963.

[43] Maurice Keller and William Marnane. Low power elliptic curve cryptography. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 310–319. 2007.

[44] Chang Han Kim, Sangho Oh, and Jongin Lim. A new hardware architecture for operations in GF(2^n). *IEEE Transactions on Computers*, 51(1):90–92, 2002.

[45] Howon Kim, Thomas Wollinger, Yongje Choi, Kyoil Chung, and Christof Paar. Hyperelliptic curve coprocessors on a FPGA. In *Workshop on Information Security Applications - WISA, Jeju Island, Korea*, pages 23–25. Springer-Verlag, 2004.

[46] N. Koblitz. *Algebraic Aspects of Cryptosystem.* Springer, 1998.

[47] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[48] Neal Koblitz. Hyperelliptic cryptosystems. *J. Cryptol.*, 1:139–150, January 1989.

[49] Unal Kocabas, Junfeng Fan, and Ingrid Verbauwhede. Implementation of Binary Edwards Curves for Very-constrained Devices. In *2010 21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP),*, pages 185 –191, july 2010.

[50] Vladislav Kovtun and Thomas Wollinger. Fast explicit formulae for genus 2 hyperelliptic curves using projective coordinates (updated), 2008. updated vladislav.kovtun@gmail.com 13911 received 2 Feb 2008.

[51] S. Kumar and C. Paar. Are Standards Compliant Elliptic Curve Cryptosystems Feasible on RFID? In *Workshop Record of the ECRYPT Workshop RFID Security*, page 19, 2006.

[52] Sandeep S. Kumar. *Elliptic Curve Cryptography for Constrained Devices.* VDM Verlag, Germany, August 2008.

[53] F. J. Kurdahi and A. C. Parker. Real: a program for register allocation. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, DAC '87, pages 210–215, New York, NY, USA, 1987.

[54] Julio López and Ricardo Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, pages 316–327, London, UK, 1999. Springer-Verlag.

[55] Jyu-Yuan Lai, Tzu-Yu Hung, Kai-Hsiang Yang, and Chih-Tsun Huang. High-performance Architecture for Elliptic Curve Cryptography Over Binary Field. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3933–3936, Paris, August 2010.

[56] T. Lange. *Efficient Arithmetic on Hyperelliptic Curves.* PhD thesis, University Gesamthochschule Essen, 2001.

[57] T. Lange. Inversion-free arithmetic on genus 2 hyperelliptic curves. Technical report, Crypology ePrint Archive, 2002.

[58] Tanja Lange. Efficient Arithmetic on Hyperelliptic Koblitz Curves. Technical report, Institute for Experimental Mathematics, 2001.

[59] Tanja Lange. Efficient Arithmetic on Genus 2 Hyperelliptic Curves over Finite Fields via Explicit Formulae. In *Cryptology ePrint archive, Report 2002/121*, 2002.

[60] Tanja Lange. Weighted coordinates on genus 2 hyperelliptic curves. Technical report, International Association for Cryptologic Research (IACR) Eprint archive, 2002. lange@itsc.ruhr-uni-bochum.de 12194 received 11 Oct 2002, last revised 22 May 2003.

[61] Tanja Lange. Formulae for arithmetic on genus 2 hyperelliptic curves. *Applicable Algebra in Engineering, Communication and Computing*, 15:295–328, 2003.

[62] Tanja Lange and Marc Stevens. Efficient doubling on genus two curves over binary fields. In *Proceedings of the 11th international conference on Selected Areas in Cryptography*, SAC'04, pages 170–181, Berlin, Heidelberg, 2005. Springer-Verlag.

[63] Weng Fook Lee. *VHDL Coding and Logic Synthesis with Synopsys.* Academic Press, 1st edition, August 2000.

[64] Yong Ki Lee, K. Sakiyama, L. Batina, and I. Verbauwhede. Elliptic-Curve-Based security processor for RFID. *IEEE Transaction on Computers*, 57(11):1514–1527, 2008.

[65] Yong Ki Lee and I. Verbauwhede. A Compact Architecture for Montgomery Elliptic Curve Scalar Multiplication Processor. *In Proc. Eighth Inteernational Workshop Infromation Security Applications (WISA'07)*, pages 115–127, 2007.

[66] K.H. Leung, K.W. Ma, W.K. Wong, and P.H.W. Leong. FPGA implementation of a microcoded elliptic curve cryptographic processor. In *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 68–76, 2000.

[67] Peng Luo, Xinan Wang, Jun Feng, and Ying Xu. Low-power Hardware Implemntation of ECC Processor Suitable for Low-cost RFID Tags. *In 9th International Conference on Solid-State and Integrated-Circuit Technology (ICSICT)*, pages 1681–1684, 2008.

[68] M. McLoone and M. Robshaw. Public key cryptography and RFID tags. In *Topics in Cryptology CT-RSA 2007*, pages 372–384. 2006.

[69] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, Inc., 1994.

[70] Victor S Miller. Use of elliptic curves in cryptography. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[71] Pradeep Kumar Mishra. Pipelined computation of scalar multiplication in elliptic curve cryptosystems (Extended version). *IEEE Trans. Comput.*, 55(8):1000–1010, 2006.

[72] Pradeep Kumar Mishra, Pinakpani Pal, and Palash Sarkar. Towards Minimizing Memory Requirement for Implementation of Hyperelliptic Curve Cryptosystems. *Springer-Verlag Berlin Heidelberg*, ISPEC 2007(LNCS 4464):269–283, 2007.

[73] Sangook Moon, Jaemin Park, and Yongsurk Lee. Fast VLSI arithmetic algorithms for high-security elliptic curve cryptographic applications. *IEEE Transactions on Consumer Electronics*, 47(3):700–708, 2001.

[74] D. Mumford. *Tata Lectures on Theta II.* Springer, 1984.

[75] National Institute of Standards and Technology (NIST). Recommended Elliptic Curve for Federal Government Use, 1999.

[76] G. Orlando and C. Paar. A super-serial galois fields multiplier for FPGAs and its application to public-key algorithms. In *Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings. Seventh Annual IEEE Symposium on*, pages 232–239, 1999.

[77] E. Öztürk and B. Sunar. Low-power elliptic curve cryptography using scaled modular arithmetic. In *Proceedings of 6th International Workshop on Cryptographic Hardware in Embedded Systems (CHES)*, pages 92–106. SpringerVerlag, 2004.

[78] Jan Pelzl. Hyperelliptic cryptosystems on embedded microprocessors. Master's thesis, Ruhr-Universit*ä*t Bochum, September 2002.

[79] Jan Pelzl, Thomas Wollinger, Jorge Guajardo, and Christof Paar. Hyperelliptic curve cryptosystems: Closing the performance gap to elliptic curves. In *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2003*, pages 351–365. Springer-Verlag, 2003.

[80] Jan Pelzl, Thomas Wollinger, and Christof Paar. High performance arithmetic for special hyperelliptic curve cryptosystems of genus two. In *International Conference on Information Technology: Coding and Computing - ITCC 2004. IEEE Computer Society*, 2004.

[81] Yong ping DAN, Xue cheng ZOU, Zheng lin LIU, Yu HAN, and Li hua YI. Design of highly efficient elliptic curve crypto-processor with two multiplications over GF(2^{163}). *The Journal of China Universities of Posts and Telecommunications*, 16(2):72–79, April 2009.

[82] Martin Christopher Rosner. Elliptic curve cryptosystems on reconfigurable hardware. *Master Thesis, Worcester Polytechnic Inst*, 1998.

[83] Anup Hosangadi Ryan Kastner and Farzan Fallah. *Arithmetic Optimization Techniques for Hardware and Software Design.* Cambridge University Press, 2010.

[84] K. Sakiyama. *Secure Design Methodology and Implementation for Embedded Public-key Cryptosystems.* PhD thesis, Katholieke Universiteit Leuven, Belgium, 2007.

[85] Richard Schroeppel, Cheryl Beaver, Rita Gonzales, Russell Miller, and Timothy Draelos. A Low-Power design for an elliptic curve digital signature chip. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 53–64. 2003.

[86] Leilei Song and Keshab K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *J. VLSI Signal Process. Syst.*, 19:149–166, July 1998.

[87] Matsuo K. Chao J. Tsujii S. Sugizaki, T. An extension of harley addition algorithm for hyperelliptic curves over finite fields of characterisitc two. Technical report, ISEC2002-9, IEICE Japan, 49-56, 2002.

[88] M. Takahashi. Improving Harley Algorithms for Jacobians of genus 2 Hyperelliptic Curves. In *Proc. of SCIS2002, IEICE Japan*, 2002. in Japanese.

[89] Pim Tuyls and Lejla Batina. RFID-tags for Anti-Counterfeiting, 2006.

[90] Frank Vahid. *Digital Design with RTL Design, Verilog and VHDL.* Wiley, 2nd edition, March 2010.

[91] J. Wolkerstorfer. Scaling ECC hardware to a minimum. *In ECRYPT workshop - Cryptographic Advances in Secure Hardware*, September 2005.

[92] T. Wollinger. Computer architectures for cryptosystems based on hyperelliptic curves. Master's thesis, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, May 2001.

[93] Thomas Wollinger. Software and hardware implementation of hyperelliptic curve cryptosystems. Technical report, Ruhr-University Bochum, Bochum, Germany, 2004.

[94] Thomas Wollinger and Christof Paar. Hardware architectures proposed for cryptosystems based on hyperelliptic curves. In *In Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems - ICECS 2002, volume III*, pages 1159–1163, 2002.

[95] Thomas Wollinger, Jan Pelzl, and Christof Paar. Cantor versus harley: Optimization and analysis of explicit formulae for hyperelliptic curve cryptosystems. *IEEE Transactions on Computers*, 54:861–872, 2005.

[96] Thomas Wollinger, Jan Pelzl, Volker Wittelsberger, Christof Paar, Gïkay Saldamli, and ïetin K. Koï. Elliptic and hyperelliptic curves on embedded μp. *ACM Transactions on Embedded Computing Systems*, 3(3):509–533, 2004.

[97] K. Matsuo J. Chao Y. Miyamoto, H. Doi and S. Tsujii. A fast addition algorithm of genus two hyperelliptic curve. In *Proc. of SCIS2002, IEICE Japan, pages 497-502*, 2002. in Japanese.

[98] Hiroto Yasuura and Hiroyuki Tomiyama. Power optimization by datapath width adjustment. In *Power Aware Design Methodologies*, pages 181–199. 2002.

[99] Xiaoyang Zeng, Xiaofang Zhou, and Qianling Zhang. Hardware/software Co-design of Elliptic Curves Public-key Cryptosystems. volume 2, pages 1496–1499, 2002.

[100] Yu Zhang, Dongdong Chen, Younhee Choi, Li Chen, and Seok-Bum Ko. A high performance pseudo-multi-core ECC processor over GF(2^{163}). In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 701–704, Paris, August 2010.

Appendix A

Efficient HECC Explicit Register Management Formulas

A.1 New Weighted Coordinates ($\mathcal{N}$)

This section contains efficient HECC explicit register management formulas specified for doubling, addition, and mixed addition in projective coordinates. These explicit formulas are the efficient ones that result after the application of register allocation, operation scheduling and storage binding procedures. These formulas are used as the basis for determining the data path and the control unit synthesis that are most useful for minimizing the hardware implementation of an HECC processor. The formulas for the projective when $h_2 = 0$ and when $h_2 \neq 0$ can be found in Algorithms A.1, A.2, A.3, A.4, A.5, and A.6.

.

A.2 Projective Coordinates ($\mathcal{P}$)

Similarly, efficient HECC explicit register management specified formulas are provided for doubling, addition, and mixed addition in projective coordinates. For counting, formulas are given for even characteristic when $h_2 = 0$ and when $h_2 \neq 0$, respectively. The formulas for the projective coordinates can be found in Algorithms A.7, A.8, A.9, A.10, A.11, and A.12.

.

Algorithm A.1 The modified register management of divisor doubling, *new weighted* coordinates in an even characteristic when $h_2 = 0$

Input: $[U_1, U_0, V_1, V_0, Z_1, Z_2, z_1, z_2, z_3, z_4]$,$h = h_1x + h_0,\ f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U_1', U_0', V_1', V_0', Z_1', Z_2', z_1', z_2', z_3', z_4'] = 2[U_1, U_0, V_1, V_0, Z_1, Z_2, z_1, z_2, z_3, z_4]$.

```
1: $R_0 \leftarrow U_1,\ R_1 \leftarrow U_0,\ R_2 \leftarrow z_1,\ R_3 \leftarrow z_4$

2: $R_4 \leftarrow cMUL\,(R_1,\ h_1)$
3: $R_5 \leftarrow cMUL\,(R_0,\ h_1)$
4: $FP(ADD, cMUL) \leftarrow ADD\,(R_4,\ R_5)$
5: $R_4 \leftarrow cMUL\,(FP(ADD, cMUL),\ h_1)$
6: $R_5 \leftarrow cMUL\,(R_2,\ SQR(h_0))$
7: $FP(ADD, MUL) \leftarrow ADD\,(R_4,\ R_5)$
8: $R_5 \leftarrow MUL\,(FP(ADD, MUL),\ R_3)$
9: $Z_2' \leftarrow MUL\,(R_5,\ R_3),\ DOUT \leftarrow Z_2'$
10: $R_4 \leftarrow cMUL\,(R_0,\ h_1)$
11: $FP(cMUL, ADD) \leftarrow cMUL\,(R_2,\ h_0)$
12: $R_6 \leftarrow ADD\,(FP(cMUL, ADD),\ R_4)$
13: $R_4 \leftarrow V_1,\ R_7 \leftarrow SQR\,(R_4)$
14: $R_8 \leftarrow SQR\,(R_0)$
15: $FP(SQR, cMUL) \leftarrow SQR\,(R_2)$
16: $R_9 \leftarrow cMUL\,(FP(SQR, cMUL),\ f_3)$
17: $FP(ADD, MUL) \leftarrow ADD\,(R_8,\ R_9)$
18: $R_8 \leftarrow MUL\,(FP(ADD, MUL),\ z_2)$
19: $R_9 \leftarrow cMUL\,(R_3,\ f_2)$
20: $FP(cMUL, ADD) \leftarrow cMUL\,(R_4,\ h_1)$
21: $R_9 \leftarrow ADD\,(FP(cMUL, ADD),\ R_9)$
22: $FP(MUL, ADD) \leftarrow MUL\,(R_3,\ R_9)$
23: $R_3 \leftarrow ADD\,(FP(MUL, ADD),\ R_7)$
24: $FP(MUL, ADD) \leftarrow MUL\,(R_0,\ R_8)$
25: $R_7 \leftarrow ADD\,(R_3,\ FP(MUL, ADD),\ R_3)$
26: $R_3 \leftarrow MUL\,(R_6,\ R_7)$
27: $R_9 \leftarrow cMUL\,(R_8,\ h_1)$
28: $R_6 \leftarrow cADD\,(R_6,\ h_1)$
29: $FP(ADD, MUL) \leftarrow ADD\,(R_7,\ R_8)$
30: $R_6 \leftarrow MUL\,(FP(ADD, MUL),\ R_6)$
31: $R_7 \leftarrow ADD\,(R_3,\ R_6)$
32: $FP(ADD, MUL) \leftarrow cADD\,(R_0,\ 1)$
33: $R_6 \leftarrow ADD\,(FP(ADD, MUL),\ R_9)$
34: $R_6 \leftarrow ADD\,(R_6,\ R_7)$
35: $R_7 \leftarrow MUL\,(R_2,\ R_9)$
36: $FP(MUL, ADD) \leftarrow MUL\,(R_7,\ R_1)$
37: $R_3 \leftarrow ADD\,(FP(MUL, ADD),\ R_3)$
38: $R_7 \leftarrow MUL\,(R_6,\ R_2)$
39: $S_0 \leftarrow SQR\,(R_3)$
40: $S \leftarrow MUL\,(R_3,\ R_7)$
41: $R_5 \leftarrow MUL\,(R_7,\ R_5)$
42: $R_2 \leftarrow SQR\,(R_7)$
43: $R_8 \leftarrow SQR\,(Z_2')$
44: $R_7 \leftarrow MUL\,(R_7,\ R_6)$
45: $R_9 \leftarrow MUL\,(R_7,\ Z_2')$
46: $z_4' \leftarrow MUL\,(R_9,\ R_2)$
47: $R_1 \leftarrow MUL\,(R_1,\ R_{13})$
48: $R_3 \leftarrow MUL\,(R_3,\ R_6)$
49: $R_{10} \leftarrow MUL\,(R_0,\ R_6)$
50: $R_7 \leftarrow MUL\,(R_1,\ R_3)$
51: $R_3 \leftarrow ADD\,(R_3,\ R_6)$
52: $FP(ADD, MUL) \leftarrow ADD\,(R_0,\ R_1)$
53: $R_3 \leftarrow MUL\,(FP(ADD, MUL),\ R_3)$
54: $R_3 \leftarrow SUB\,(R_3,\ R_7)$
55: $R_3 \leftarrow SUB\,(R_3,\ R_{10})$
56: $R_{10} \leftarrow ADD\,(R_{10},\ S)$
57: $FP(cMUL, ADD) \leftarrow cMUL\,(R_9,\ h_1)$
58: $R_0 \leftarrow ADD\,(FP(cMUL, ADD),\ S_0)$
59: $R_{10} \leftarrow SUB\,(R_{10},\ R_8)$
60: $R_1 \leftarrow MUL\,(R_{10},\ R_0)$
61: $R_6 \leftarrow MUL\,(R_{10},\ R_8)$
62: $FP(MUL, ADD) \leftarrow MUL\,(R_5,\ R_4)$
63: $R_4 \leftarrow ADD\,(FP(MUL, ADD),\ R_3)$
64: $FP(ADD, MUL) \leftarrow ADD\,(R_0,\ R_4)$
65: $R_0 \leftarrow MUL\,(FP(ADD, MUL),\ R_2)$
66: $R_0 \leftarrow ADD\,(R_0,\ R_6)$
67: $FP(cMUL, ADD) \leftarrow cMUL\,(R_9,\ h_1)$
68: $V_1' \leftarrow ADD\,(FP(cMUL, ADD),\ R_0)$
69: $FP(MUL, ADD) \leftarrow MUL\,(R_5,\ V_0)$
70: $R_0 \leftarrow ADD\,(FP(MUL, ADD),\ R_7)$
71: $FP(MUL, ADD) \leftarrow MUL\,(R_0,\ R_2)$
72: $R_0 \leftarrow ADD\,(FP(MUL, ADD),\ R_1)$
73: $FP(cMUL, ADD) \leftarrow cMUL\,(h_0,\ R_9)$
74: $V_0' \leftarrow ADD\,(FP(cMUL, ADD),\ R_0)$
```

Algorithm A.2 The modified register management of divisor addition, *new weighted* coordinates in an even characteristic when $h_2 = 0$

```
Input: $[U_{11}, U_{10}, V_{11}, V_{10}, Z_{11}, Z_{12}, z_{11}, z_{12}, z_{13}, z_{14}]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}]$,
       $h = h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U'_1, U'_0, V'_1, V'_0, Z'_1, Z'_2, z'_1, z'_2, z'_3, z'_4]$
 1: $R_0 \leftarrow U_{11},\ R_1 \leftarrow U_{10}, R_2 \leftarrow V_{11},\ R_3 \leftarrow V_{10}, R_4 \leftarrow z_{11},\ R_5 \leftarrow z_{13}$
 2: $R_6 \leftarrow z_{14},\ R_7 \leftarrow V_{21},\ R_8 \leftarrow V_{20},\ R_9 \leftarrow z_{21},\ R_{10} \leftarrow z_{23}$

 3: $R_{11} \leftarrow MUL\,(R_4,\ U_{21})$
 4: $R_{12} \leftarrow MUL\,(R_4,\ U_{20})$
 5: $R_7 \leftarrow MUL\,(R_6,\ R_7)$
 6: $FP(MUL,\ ADD) \leftarrow MUL\,(R_0,\ R_9)$
 7: $R_{10} \leftarrow MUL\,(FP(MUL,\ ADD),\ R_{11})$
 8: $FP(MUL,\ ADD) \leftarrow MUL\,(R_1,\ R_9)$
 9: $R_9 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{12})$
10: $R_{13} \leftarrow MUL\,(R_0,\ R_{10})$
11: $FP(MUL,\ ADD) \leftarrow MUL\,(R_6,\ R_4)$
12: $R_{13} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{13})$
13: $FP(SQR,\ MUL) \leftarrow SQR\,(R_{10})$
14: $R_{14} \leftarrow ADD\,(FP(SQR,\ MUL),\ R_1)$
15: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{13},\ R_9)$
16: $R_{14} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{14})$
17: $R_5 \leftarrow MUL\,(R_{14},\ R_5)$
18: $Z'_2 \leftarrow ADD\,(R_5,\ R_6)$
19: $FP(SQR,\ MUL) \leftarrow SQR\,(R_5)$
20: $R_5 \leftarrow MUL\,(FP(SQR,\ MUL),\ R_6)$
21: $FP(MUL,\ ADD) \leftarrow MUL\,(R_3,\ z_{24})$
22: $R_3 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_8)$
23: $R_{15} \leftarrow MUL\,(R_3,\ R_{13})$
24: $R_{16} \leftarrow MUL\,(R_2,\ R_{10})$
25: $FP(ADD,\ MUL) \leftarrow MUL\,(R_0,\ R_4)$
26: $R_0 \leftarrow MUL\,(FP(ADD,\ MUL)\,R_{16})$
27: $R_0 \leftarrow ADD\,(R_0,\ R_{15})$
28: $R_4 \leftarrow MUL\,(R_4,\ R_{10})$
29: $FP(ADD,\ MUL) \leftarrow ADD\,(R_4,\ R_{13})$
30: $R_2 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_2)$
31: $FP(MUL,\ ADD) \leftarrow MUL\,(R_1,\ R_{16})$
32: $R_{15} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{15})$
33: $R_2 \leftarrow MUL\,(R_0,\ R_6)$
34: $R_3 \leftarrow MUL\,(R_2,\ R_{14})$
35: $R_6 \leftarrow MUL\,(R_{15},\ R_6)$
36: $R_4 \leftarrow MUL\,(R_2,\ R_6)$
37: $R_{13} \leftarrow SQR\,(Z'_2)$
38: $R_{14} \leftarrow MUL\,(R_2,\ Z'_2)$
39: $z'_4 \leftarrow MUL\,(R_{14},\ R_{16})$
40: $R_2 \leftarrow ADD\,(R_2,\ R_0)$
41: $FP(ADD,\ MUL) \leftarrow ADD\,(R_{17},\ U_{21})$
42: $R_2 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_2)$
43: $FP(ADD,\ ADD) \leftarrow ADD\,(R_2,\ R_{15})$
44: $R_2 \leftarrow ADD\,(FP(ADD,\ ADD),\ R_{12})$
45: $FP(ADD,\ MUL) \leftarrow ADD\,(R_{10},\ R_{11})$
46: $R_1 \leftarrow ADD\,(FP(ADD,\ MUL),\ R_1)$
47: $FP(ADD,\ MUL) \leftarrow ADD\,(R_1,\ R_5)$
48: $R_1 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{10})$
49: $R_9 \leftarrow MUL\,(R_0,\ R_9)$
50: $R_6 \leftarrow ADD\,(R_6,\ R_9)$
51: $FP(MUL,\ ADD) \leftarrow MUL\,(h_1,\ R_{14})$
52: $R_6 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_6)$
53: $FP(MUL,\ ADD) \leftarrow MUL\,(R_0,\ R_{10})$
54: $R_{13} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{13})$
55: $FP(MUL,\ ADD) \leftarrow MUL\,(R_3,\ R_7)$
56: $R_2 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_2)$
57: $FP(ADD,\ MUL) \leftarrow ADD\,(R_2,\ R_6)$
58: $R_2 \leftarrow ADD\,(FP(ADD,\ MUL),\ R_1)$
59: $FP(MUL,\ ADD) \leftarrow MUL\,(h_1,\ R_2)$
60: $R_4 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_4)$
61: $FP(MUL,\ ADD) \leftarrow MUL\,(R_3,\ R_8)$
62: $R_{12} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{12})$
```

Algorithm A.3 The modified register management of mixed addition, *new weighted* coordinates in an even characteristic when $h_2 = 0$

```
Input: $[U_{11}, U_{10}, V_{11}, V_{10}]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}]$,
       $h = h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U'_1, U'_0, V'_1, V'_0, Z'_1, Z'_2, z'_1, z'_2, z'_3, z'_4]$
 1: $R_0 \leftarrow V_{21}$, $R_1 \leftarrow V_{20}$,$R_2 \leftarrow U_{21}$, $R_3 \leftarrow U_{20}$,$R_4 \leftarrow Z_{21}$, $R_5 \leftarrow z_{21}$
 2: $R_6 \leftarrow z_{23}$, $R_7 \leftarrow z_{24}$

 3: $FP(MUL, ADD) \leftarrow MUL(R_5, U_{11})$
 4: $R_8 \leftarrow ADD(FP(MUL, ADD), R_2)$
 5: $FP(MUL, ADD) \leftarrow MUL(R_5, U_{10})$
 6: $R_5 \leftarrow ADD(FP(MUL, ADD), R_3)$
 7: $FP(MUL, ADD) \leftarrow MUL(R_8, U_{11})$
 8: $R_9 \leftarrow ADD(FP(MUL, ADD), R_5)$
 9: $FP(SQR, MUL) \leftarrow SQR(R_8)$
10: $R_{10} \leftarrow MUL(FP(SQR, MUL), U_{10})$
11: $FP(MUL, ADD) \leftarrow MUL(R_5, R_9)$
12: $R_{10} \leftarrow ADD(FP(MUL, ADD), R_{10})$
13: $FP(MUL, ADD) \leftarrow MUL(R_7, V_{10})$
14: $R_{11} \leftarrow ADD(FP(MUL, ADD), R_1)$
15: $FP(MUL, ADD) \leftarrow MUL(R_7, V_{11})$
16: $R_7 \leftarrow ADD(FP(MUL, ADD), V_{21})$
17: $R_{15} \leftarrow MUL(FP(ADD, MUL), R_{13})$
18: $FP(ADD, MUL) \leftarrow ADD(R_7, R_{11})$
19: $R_{14} \leftarrow MUL(FP(ADD, MUL), R_{14})$
20: $FP(SUB, SUB) \leftarrow SUB(R_{12}, R_{14})$
21: $R_{14} \leftarrow SUB(FP(SUB, SUB), R_{15})$
22: $R_{14} \leftarrow MUL(R_{14}, R_{10})$
23: $FP(MUL, ADD) \leftarrow MUL(U_{10} R_{13})$
24: $R_9 \leftarrow ADD(FP(MUL, ADD), R_{12})$
25: $R_7 \leftarrow SQR(R_7)$
26: $R_9 \leftarrow MUL(R_{10}, R_{14})$
27: $R_{11} \leftarrow MUL(Z'_2, R_{11})$
28: $z'_4 \leftarrow MUL(R_{14}, R_{11})$
29: $R_{13} \leftarrow MUL(R_{12}, R_4)$
30: $R_9 \leftarrow MUL(R_9, R_3)$
31: $FP(ADD, MUL) \leftarrow ADD(R_2, R_3)$
32: $R_2 \leftarrow MUL(FP(ADD, MUL), R_{15})$
33: $FP(SUB, SUB) \leftarrow SUB(R_2, R_9)$
34: $R_2 \leftarrow SUB(FP(SUB, SUB), R_4)$
35: $R_4 \leftarrow ADD(R_4, R_{13})$
36: $R_3 \leftarrow MUL(U_{11}, R_7)$
37: $FP(MUL, ADD) \leftarrow ADD(R_6, R_3)$
38: $R_6 \leftarrow MUL(FP(MUL, ADD), R_8)$
39: $FP(MUL, ADD) \leftarrow MUL(R_7, R_8)$
40: $R_5 \leftarrow ADD(FP(MUL, ADD), z'_2)$
41: $R_4 \leftarrow ADD(R_4, R_5)$
42: $R_6 \leftarrow MUL(R_3, R_4)$
43: $FP(MUL, ADD) \leftarrow MUL(R_0, R_{12})$
44: $R_2 \leftarrow ADD(FP(MUL, ADD), R_2)$
45: $R_4 \leftarrow ADD(R_4, R_2)$
46: $FP(ADD, MUL) \leftarrow ADD(R_3, R_4)$
47: $R_2 \leftarrow ADD(FP(ADD, MUL), R_{14})$
48: $R_0 \leftarrow ADD(R_2, R_6)$
49: $FP(MUL, ADD) \leftarrow MUL(R_1, R_{12})$
50: $R_4 \leftarrow ADD(FP(MUL, ADD), R_9)$
51: $FP(ADD, MUL) \leftarrow ADD(R_2, R_4)$
52: $R_4 \leftarrow MUL(FP(ADD, MUL), R_{14})$
```

Algorithm A.4 The modified register management of divisor doubling , *new weighted* coordinates in an even characteristic when $h_2 \neq 0$

```
Input: $[U_1, U_0, V_1, V_0, Z_1, Z_2, z_1, z_2, z_3, z_4]$, $h = h_2x^2+h_1x+h_0$, $f = x^5+f_3x^3+f_2x^2+f_1x+f_0$.
Output: $[U_1', U_0', V_1', V_0', Z_1', Z_2', z_1', z_2', z_3', z_4'] = 2[U_1, U_0, V_1, V_0, Z_1, Z_2, z_1, z_2, z_3, z_4]$.
1: $R_0 \leftarrow U_1,\ R_1 \leftarrow U_0,\ R_2 \leftarrow z_1,\ R_3 \leftarrow z_4$

2: $FP(ADD, cMUL) \leftarrow ADD\,(R_4,\ R_5)$
3: $R_4 \leftarrow cMUL\,(FP(ADD, cMUL),\ h_1)$
4: $R_5 \leftarrow cMUL\,(R_2,\ SQR(h_0))$
5: $FP(ADD, MUL) \leftarrow ADD\,(R_4,\ R_5)$
6: $R_5 \leftarrow MUL\,(FP(ADD, MUL),\ R_3)$
7: $Z_2' \leftarrow MUL\,(R_5,\ R_3),\ DOUT \leftarrow Z_2'$
8: $FP(cMUL, ADD) \leftarrow cMUL\,(R_2,\ h_0)$
9: $R_6 \leftarrow ADD\,(FP(cMUL, ADD),\ R_4)$
10: $R_8 \leftarrow SQR\,(R_0)$
11: $FP(SQR, cMUL) \leftarrow SQR\,(R_2)$
12: $R_9 \leftarrow cMUL\,(FP(SQR, cMUL),\ f_3)$
13: $FP(ADD, MUL) \leftarrow ADD\,(R_8,\ R_9)$
14: $R_8 \leftarrow MUL\,(FP(ADD, MUL),\ z_2)$
15: $FP(cMUL, ADD) \leftarrow cMUL\,(R_4,\ h_1)$
16: $R_9 \leftarrow ADD\,(FP(cMUL, ADD),\ R_9)$
17: $FP(MUL, ADD) \leftarrow MUL\,(R_3,\ R_9)$
18: $R_3 \leftarrow ADD\,(FP(MUL, ADD),\ R_7)$
19: $FP(MUL, ADD) \leftarrow MUL\,(R_0,\ R_8)$
20: $R_7 \leftarrow ADD\,(R_3,\ FP(MUL, ADD),\ R_3)$
21: $R_3 \leftarrow MUL\,(R_6,\ R_7)$
22: $R_9 \leftarrow cMUL\,(R_8,\ h_1)$
23: $R_6 \leftarrow cADD\,(R_6,\ h_1)$
24: $FP(ADD, MUL) \leftarrow ADD\,(R_7,\ R_8)$
25: $R_6 \leftarrow MUL\,(FP(ADD, MUL),\ R_6)$
26: $R_7 \leftarrow ADD\,(R_3,\ R_6)$
27: $FP(ADD, MUL) \leftarrow cADD\,(R_0,\ 1)$
28: $R_6 \leftarrow ADD\,(FP(ADD, MUL),\ R_9)$
29: $R_6 \leftarrow ADD\,(R_6,\ R_7)$
30: $R_7 \leftarrow MUL\,(R_2,\ R_9)$
31: $FP(MUL, ADD) \leftarrow MUL\,(R_7,\ R_1)$
32: $R_3 \leftarrow ADD\,(FP(MUL, ADD),\ R_3)$
33: $R_7 \leftarrow MUL\,(R_6,\ R_2)$
34: $S_0 \leftarrow SQR\,(R_3)$
35: $S \leftarrow MUL\,(R_3,\ R_7)$
36: $R_5 \leftarrow MUL\,(R_7,\ R_5)$
37: $R_2 \leftarrow SQR\,(R_7)$
38: $R_8 \leftarrow SQR\,(Z_2')$
39: $R_7 \leftarrow MUL\,(R_7,\ R_6)$
40: $R_9 \leftarrow MUL\,(R_7,\ Z_2')$
41: $z_4' \leftarrow MUL\,(R_9,\ R_2)$
42: $R_1 \leftarrow MUL\,(R_1,\ R_{13})$
43: $R_3 \leftarrow MUL\,(R_3,\ R_6)$
44: $R_{10} \leftarrow MUL\,(R_0,\ R_6)$
45: $R_7 \leftarrow MUL\,(R_1,\ R_3)$
46: $R_3 \leftarrow ADD\,(R_3,\ R_6)$
47: $FP(ADD, MUL) \leftarrow ADD\,(R_0,\ R_1)$
48: $R_3 \leftarrow MUL\,(FP(ADD, MUL),\ R_3)$
49: $R_3 \leftarrow SUB\,(R_3,\ R_7)$
50: $R_3 \leftarrow SUB\,(R_3,\ R_{10})$
51: $R_{10} \leftarrow ADD\,(R_{10},\ S)$
52: $FP(cMUL, ADD) \leftarrow cMUL\,(R_9,\ h_1)$
53: $R_0 \leftarrow ADD\,(FP(cMUL, ADD),\ S_0)$
54: $R_{10} \leftarrow SUB\,(R_{10},\ R_8)$
55: $R_1 \leftarrow MUL\,(R_{10},\ R_0)$
56: $R_6 \leftarrow MUL\,(R_{10},\ R_8)$
57: $FP(MUL, ADD) \leftarrow MUL\,(R_5,\ R_4)$
58: $R_4 \leftarrow ADD\,(FP(MUL, ADD),\ R_3)$
59: $FP(ADD, MUL) \leftarrow ADD\,(R_0,\ R_4)$
60: $R_0 \leftarrow MUL\,(FP(ADD, MUL),\ R_2)$
61: $R_0 \leftarrow ADD\,(R_0,\ R_6)$
62: $FP(cMUL, ADD) \leftarrow cMUL\,(R_9,\ h_1)$
63: $V_1' \leftarrow ADD\,(FP(cMUL, ADD),\ R_0)$
64: $FP(MUL, ADD) \leftarrow MUL\,(R_5,\ V_0)$
65: $R_0 \leftarrow ADD\,(FP(MUL, ADD),\ R_7)$
66: $FP(MUL, ADD) \leftarrow MUL\,(R_0,\ R_2)$
67: $R_0 \leftarrow ADD\,(FP(MUL, ADD),\ R_1)$
68: $FP(cMUL, ADD) \leftarrow cMUL\,(h_0,\ R_9)$
69: $V_0' \leftarrow ADD\,(FP(cMUL, ADD),\ R_0)$
```

Algorithm A.5 The modified register management of divisor addition, *new weighted* coordinates in an even characteristic when $h_2 \neq 0$

Input: $[U_{11}, U_{10}, V_{11}, V_{10}, Z_{11}, Z_{12}, z_{11}, z_{12}, z_{13}, z_{14}]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}]$, $h = h_2x^2 + h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.

Output: $[U'_1, U'_0, V'_1, V'_0, Z'_1, Z'_2, z'_1, z'_2, z'_3, z'_4]$

```
1: $R_0 \leftarrow V_{21}$, $R_1 \leftarrow V_{20}$,$R_2 \leftarrow U_{21}$, $R_3 \leftarrow U_{20}$,$R_4 \leftarrow Z_{21}$, $R_5 \leftarrow z_{21}$
2: $R_6 \leftarrow z_{23}$, $R_7 \leftarrow z_{24}$

3: $FP(MUL, ADD) \leftarrow MUL(R_5, U_{11})$
4: $R_8 \leftarrow ADD(FP(MUL, ADD), R_2)$
5: $FP(MUL, ADD) \leftarrow MUL(R_5, U_{10})$
6: $R_5 \leftarrow ADD(FP(MUL, ADD), R_3)$
7: $FP(MUL, ADD) \leftarrow MUL(R_8, U_{11})$
8: $R_9 \leftarrow ADD(FP(MUL, ADD), R_5)$
9: $FP(SQR, MUL) \leftarrow SQR(R_8)$
10: $R_{10} \leftarrow MUL(FP(SQR, MUL), U_{10})$
11: $FP(MUL, ADD) \leftarrow MUL(R_5, R_9)$
12: $R_{10} \leftarrow ADD(FP(MUL, ADD), R_{10})$
13: $FP(MUL, ADD) \leftarrow MUL(R_7, V_{10})$
14: $R_{11} \leftarrow ADD(FP(MUL, ADD), R_1)$
15: $FP(MUL, ADD) \leftarrow MUL(R_7, V_{11})$
16: $R_7 \leftarrow ADD(FP(MUL, ADD), V_{21})$
17: $R_{13} \leftarrow MUL(R_7, R_8)$
18: $FP(ADD, MUL) \leftarrow ADD(U_{11}, 1)$
19: $R_{15} \leftarrow MUL(FP(ADD, MUL), R_{13})$
20: $R_{14} \leftarrow ADD(R_8, R_9)$
21: $FP(ADD, MUL) \leftarrow ADD(R_7, R_{11})$
22: $R_{14} \leftarrow MUL(FP(ADD, MUL), R_{14})$
23: $FP(SUB, SUB) \leftarrow SUB(R_{12}, R_{14})$
24: $R_{14} \leftarrow SUB(FP(SUB, SUB), R_{15})$
25: $R_{14} \leftarrow MUL(R_{14}, R_{10})$
26: $FP(MUL, ADD) \leftarrow MUL(U_{10} R_{13})$
27: $R_9 \leftarrow ADD(FP(MUL, ADD), R_{12})$
28: $R_{11} \leftarrow MUL(R_7, R_4)$
29: $R_{12} \leftarrow MUL(R_7, R_{10})$
30: $R_{10} \leftarrow MUL(R_4, R_9)$
31: $R_{13} \leftarrow MUL(R_{10}, R_{11})$
32: $R_{10} \leftarrow SQR(R_{10})$
33: $R_9 \leftarrow MUL(R_7, R_9)$
34: $R_7 \leftarrow SQR(R_7)$
35: $R_9 \leftarrow MUL(R_{10}, R_{14})$
36: $R_{14} \leftarrow SQR(R_{11})$
37: $z'_2 \leftarrow SQR(Z'_2)$
38: $R_{11} \leftarrow MUL(Z'_2, R_{11})$
39: $FP(ADD, MUL) \leftarrow ADD(R_2, R_3)$
40: $R_2 \leftarrow MUL(FP(ADD, MUL), R_{15})$
41: $FP(SUB, SUB) \leftarrow SUB(R_2, R_9)$
42: $R_2 \leftarrow SUB(FP(SUB, SUB), R_4)$
43: $R_4 \leftarrow ADD(R_4, R_{13})$
44: $R_3 \leftarrow MUL(U_{11}, R_7)$
45: $FP(MUL, ADD) \leftarrow ADD(R_6, R_3)$
46: $R_6 \leftarrow MUL(FP(MUL, ADD), R_8)$
47: $R_3 \leftarrow ADD(R_{10}, R_6)$
48: $R_5 \leftarrow MUL(R_5, R_7)$
49: $R_6 \leftarrow cMUL(h_1, R_{11})$
50: $R_5 \leftarrow ADD(R_5, R_3)$
51: $R_3 \leftarrow ADD(R_5, R_6)$
52: $FP(MUL, ADD) \leftarrow MUL(R_7, R_8)$
53: $R_5 \leftarrow ADD(FP(MUL, ADD), z'_2)$
54: $R_4 \leftarrow ADD(R_4, R_5)$
55: $R_6 \leftarrow MUL(R_3, R_4)$
56: $R_7 \leftarrow MUL(R_4, R_5)$
57: $FP(MUL, ADD) \leftarrow MUL(R_0, R_{12})$
58: $R_2 \leftarrow ADD(FP(MUL, ADD), R_2)$
59: $FP(ADD, MUL) \leftarrow ADD(R_3, R_4)$
60: $R_2 \leftarrow ADD(FP(ADD, MUL), R_{14})$
61: $R_0 \leftarrow ADD(R_2, R_6)$
62: $FP(MUL, ADD) \leftarrow MUL(R_1, R_{12})$
63: $R_4 \leftarrow ADD(FP(MUL, ADD), R_9)$
64: $FP(ADD, MUL) \leftarrow ADD(R_2, R_4)$
65: $R_4 \leftarrow MUL(FP(ADD, MUL), R_{14})$
```

Algorithm A.6 The modified register management of mixed addition for *new weighted* coordinates for an even characteristic when $h_2 \neq 0$

```
Input: $[U_{11}, U_{10}, V_{11}, V_{10}]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_{21}, Z_{22}, z_{21}, z_{22}, z_{23}, z_{24}]$,
       $h = h_2x^2 + h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U'_1, U'_0, V'_1, V'_0, Z'_1, Z'_2, z'_1, z'_2, z'_3, z'_4]$
 1: $R_0 \leftarrow V_{21},\ R_1 \leftarrow V_{20}$,$R_2 \leftarrow U_{21},\ R_3 \leftarrow U_{20}$,$R_4 \leftarrow Z_{21},\ R_5 \leftarrow z_{21}$
 2: $R_6 \leftarrow z_{23},\ R_7 \leftarrow z_{24}$

 3: $FP(MUL,\ ADD) \leftarrow MUL\,(R_5,\ U_{11})$
 4: $R_8 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_2)$
 5: $FP(MUL,\ ADD) \leftarrow MUL\,(R_5,\ U_{10})$
 6: $R_5 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_3)$
 7: $FP(MUL,\ ADD) \leftarrow MUL\,(R_8,\ U_{11})$
 8: $R_9 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_5)$
 9: $FP(SQR,\ MUL) \leftarrow SQR\,(R_8)$
10: $R_{10} \leftarrow MUL\,(FP(SQR,\ MUL),\ U_{10})$
11: $FP(MUL,\ ADD) \leftarrow MUL\,(R_5,\ R_9)$
12: $R_{10} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{10})$
13: $R_6 \leftarrow MUL\,(R_6,\ R_{10})$
14: $Z'_2 \leftarrow MUL\,(R_6,\ R_4)$
15: $R_6 \leftarrow SQR\,(R_6)$
16: $FP(MUL,\ ADD) \leftarrow MUL\,(R_7,\ V_{10})$
17: $R_{11} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_1)$
18: $FP(MUL,\ ADD) \leftarrow MUL\,(R_7,\ V_{11})$
19: $R_7 \leftarrow ADD\,(FP(MUL,\ ADD),\ V_{21})$
20: $R_{13} \leftarrow MUL\,(R_7,\ R_8)$
21: $FP(ADD,\ MUL) \leftarrow ADD\,(U_{11},\ 1)$
22: $R_{15} \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{13})$
23: $R_{14} \leftarrow ADD\,(R_8,\ R_9)$
24: $FP(ADD,\ MUL) \leftarrow ADD\,(R_7,\ R_{11})$
25: $R_{14} \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{14})$
26: $FP(SUB,\ SUB) \leftarrow SUB\,(R_{12},\ R_{14})$
27: $R_{14} \leftarrow SUB\,(FP(SUB,\ SUB),\ R_{15})$
28: $R_{14} \leftarrow MUL\,(R_{14},\ R_{10})$
29: $FP(MUL,\ ADD) \leftarrow MUL\,(U_{10}\,R_{13})$
30: $R_9 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{12})$
31: $R_{11} \leftarrow MUL\,(R_7,\ R_4)$
32: $R_{12} \leftarrow MUL\,(R_7,\ R_{10})$
33: $R_{10} \leftarrow MUL\,(R_4,\ R_9)$
34: $R_{13} \leftarrow MUL\,(R_{10},\ R_{11})$
35: $R_{10} \leftarrow SQR\,(R_{10})$
36: $R_9 \leftarrow MUL\,(R_7,\ R_9)$
37: $R_7 \leftarrow SQR\,(R_7)$
38: $R_9 \leftarrow MUL\,(R_{10},\ R_{14})$
39: $R_{14} \leftarrow SQR\,(R_{11})$
40: $z'_2 \leftarrow SQR\,(Z'_2)$
41: $R_{11} \leftarrow MUL\,(Z'_2,\ R_{11})$
42: $FP(ADD,\ MUL) \leftarrow ADD\,(R_2,\ R_3)$
43: $R_2 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{15})$
44: $FP(MUL,\ ADD) \leftarrow ADD\,(R_6,\ R_3)$
45: $R_6 \leftarrow MUL\,(FP(MUL,\ ADD),\ R_8)$
46: $R_3 \leftarrow ADD\,(R_{10},\ R_6)$
47: $R_5 \leftarrow MUL\,(R_5,\ R_7)$
48: $R_6 \leftarrow cMUL\,(h_1,\ R_{11})$
49: $R_5 \leftarrow ADD\,(R_5,\ R_3)$
50: $R_3 \leftarrow ADD\,(R_5,\ R_6)$
51: $FP(MUL,\ ADD) \leftarrow MUL\,(R_7,\ R_8)$
52: $R_5 \leftarrow ADD\,(FP(MUL,\ ADD),\ z'_2)$
53: $R_4 \leftarrow ADD\,(R_4,\ R_5)$
54: $R_6 \leftarrow MUL\,(R_3,\ R_4)$
55: $R_7 \leftarrow MUL\,(R_4,\ R_5)$
56: $R_4 \leftarrow cMUL\,(h_1,\ R_{11})$
57: $FP(MUL,\ ADD) \leftarrow MUL\,(R_0,\ R_{12})$
58: $R_2 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_2)$
59: $R_4 \leftarrow ADD\,(R_4,\ R_2)$
60: $FP(ADD,\ MUL) \leftarrow ADD\,(R_3,\ R_4)$
61: $R_2 \leftarrow ADD\,(FP(ADD, MUL),\ R_{14})$
62: $R_0 \leftarrow ADD\,(R_2,\ R_6)$
63: $R_2 \leftarrow cMUL\,(h_1,\ R_{11})$
64: $FP(MUL,\ ADD) \leftarrow MUL\,(R_1,\ R_{12})$
65: $R_4 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_9)$
66: $FP(ADD,\ MUL) \leftarrow ADD\,(R_2,\ R_4)$
67: $R_4 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{14})$
68: $R_1 \leftarrow ADD\,(R_6,\ R_4)$
```

Algorithm A.7 The modified register management of the divisor doubling for *projective* coordinate for an even characteristic when $h_2 = 0$

Input: $[U_1, U_0, V_1, V_0, Z]$, $h = h_1x + h_0$, $f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.

Output: $[U_1', U_0', V_1', V_0', Z'] = 2[U_1, U_0, V_1, V_0, Z]$.

```
1: $R_0 \leftarrow U_1,\ R_1 \leftarrow U_0, R_2 \leftarrow V_1,\ R_3 \leftarrow V_0, R_4 \leftarrow Z$

2: $R_4 \leftarrow SQR\,(Z)$
3: $R_5 \leftarrow SQR\,(R_2)$
4: $R_3 \leftarrow SQR\,(R_0,\ U_1)$
5: $FP(SQR,\ MUL) \leftarrow SQR\,(R_4)$
6: $R_4 \leftarrow ADD\,(FP(SQR,\ MUL),\ R_1)$
7: $R_7 \leftarrow SQR\,(R_0)$
8: $R_8 \leftarrow MUL\,(R_0,\ R_7)$
9: $FP(MUL,\ ADD) \leftarrow MUL\,(R_2,\ R_3)$
10: $R_5 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_5)$
11: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{10},\ R_9)$
12: $R_5 \leftarrow MUL\,(FP(MUL,\ ADD),\ R_8)$
13: $R_8 \leftarrow MUL\,(R_5,\ R_6)$
14: $R_2 \leftarrow MUL\,(R_3,\ R_7)$
15: $R_9 \leftarrow MUL\,(R_1,\ R_3)$
16: $FP(MUL,\ ADD) \leftarrow MUL\,(R_2,\ R_9)$
17: $R_9 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_8)$
18: $R_6 \leftarrow ADD\,(R_3,\ R_6)$
19: $FP(ADD,\ MUL) \leftarrow ADD\,(R_5,\ R_7)$
20: $R_5 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_6)$
21: $R_5 \leftarrow ADD\,(R_5,\ R_8)$
22: $FP(ADD,\ MUL) \leftarrow ADD\,(1,\ R_0)$
23: $R_6 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_2)$
24: $R_5 \leftarrow ADD\,(R_5,\ R_6)$
25: $R_6 \leftarrow MUL\,(R_3,\ R_5)$
26: $R_7 \leftarrow MUL\,(R_4,\ R_6)$
27: $R_8 \leftarrow ADD\,(R_5,\ R_6)$
28: $R_2 \leftarrow MUL\,(R_5,\ R_9)$
29: $R_3 \leftarrow MUL\,(R_2,\ R_3)$
30: $R_5 \leftarrow ADD\,(R_4,\ R_5)$
31: $R_0 \leftarrow MUL\,(R_0,\ R_8)$
32: $R_8 \leftarrow ADD\,(R_2,\ R_8)$
33: $R_2 \leftarrow MUL\,(R_1,\ R_2)$
34: $FP(ADD,\ MUL) \leftarrow ADD\,(R_1,\ U_1)$
35: $R_1 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_8)$
36: $R_1 \leftarrow ADD\,(R_1,\ R_2)$
37: $R_9 \leftarrow SQR\,(R_9)$
38: $R_1 \leftarrow ADD\,(R_0,\ R_1)$
39: $R_9 \leftarrow ADD\,(R_7,\ R_9)$
40: $R_4 \leftarrow SQR\,(R_4)$
41: $U_0' \leftarrow MUL\,(R_7,\ R_9)$
42: $U_1' \leftarrow MUL\,(R_4,\ R_7)$
43: $R_6 \leftarrow SQR\,(R_6)$
44: $Z' \leftarrow MUL\,(R_7,\ R_6)$
45: $R_8 \leftarrow ADD\,(R_0,\ R_4)$
46: $FP(ADD,\ MUL) \leftarrow ADD\,(R_3,\ R_8)$
47: $R_8 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_9)$
48: $FP(MUL,\ ADD) \leftarrow MUL\,(R_5,\ V_0)$
49: $R_2 \leftarrow SUB\,(FP(MUL,\ ADD),\ R_2)$
50: $FP(MUL,\ ADD) \leftarrow MUL\,(R_2,\ R_6)$
51: $V_0' \leftarrow ADD\,(FP(MUL,\ ADD),\ R_8)$
52: $R_0 \leftarrow ADD\,(R_0,\ R_4)$
53: $FP(ADD,\ MUL) \leftarrow ADD\,(R_0,\ R_3)$
54: $R_4 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_4)$
55: $FP(MUL,\ ADD) \leftarrow MUL\,(R_5,\ V_1)$
56: $R_1 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_1)$
57: $R_7 \leftarrow ADD\,(R_1,\ R_7)$
58: $FP(ADD,\ MUL) \leftarrow ADD\,(R_7,\ R_9)$
59: $R_6 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_6)$
60: $V_1' \leftarrow ADD\,(R_4,\ R_6)$
```

Algorithm A.8 The modified register management for divisor addition for *projective* coordinates for an even characteristic when $h_2 = 0$

```
Input: $[U_{11}, U_{10}, V_{11}, V_{10}, Z_1]$; $[U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$;$h = h_1x + h_0$, $f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U'_1, U'_0, V'_1, V'_0, Z'] = [U_{11}, U_{10}, V_{11}, V_{10}, Z_1] + [U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$
1: $R_0 \leftarrow U_{11}$, $R_1 \leftarrow U_{10}$, $R_2 \leftarrow Z_1$, $R_3 \leftarrow Z_2$

2: $R_4 \leftarrow MUL(R_2, U_{21})$
3: $\tilde{U}_{20} \leftarrow MUL(R_2, U_{20})$
4: $\tilde{V}_{21} \leftarrow MUL(R_2, V_{21})$
5: $V'_{20} \leftarrow MUL(R_2, V_{20})$
6: $FP(MUL, ADD) \leftarrow MUL(R_0, R_3)$
7: $R_5 \leftarrow ADD(FP(MUL, ADD), R_4)$
8: $FP(MUL, ADD) \leftarrow MUL(R_1, R_3)$
9: $R_6 \leftarrow MUL(FP(MUL, ADD), \tilde{U}_{20})$
10: $R_7 \leftarrow MUL(R_0, R_5)$
11: $FP(MUL, ADD) \leftarrow MUL(R_2, R_6)$
12: $R_7 \leftarrow ADD(FP(MUL, ADD), R_7)$
13: $R_8 \leftarrow MUL(R_6, R_7)$
14: $FP(SQR, MUL) \leftarrow SQR(R_5)$
15: $R_9 \leftarrow MUL(FP(SQR, MUL), R_1)$
16: $R_8 \leftarrow ADD(R_8, R_9)$
17: $FP(MUL, ADD) \leftarrow MUL(R_3, V_{10})$
18: $R_9 \leftarrow ADD(FP(MUL, ADD), \tilde{V}_{20})$
19: $R_{10} \leftarrow MUL(R_7, R_9)$
20: $FP(MUL, ADD) \leftarrow MUL(R_2, R_5)$
21: $R_7 \leftarrow ADD(FP(MUL, ADD), R_7)$
22: $FP(MUL, ADD) \leftarrow MUL(R_3, V_{11})$
23: $R_{11} \leftarrow ADD(FP(MUL, ADD), \tilde{V}_{21})$
24: $FP(ADD, MUL) \leftarrow ADD(R_9, R_{11})$
25: $R_7 \leftarrow MUL(FP(ADD, MUL), R_7)$
26: $R_9 \leftarrow MUL(R_5, R_{11})$
27: $R_9 \leftarrow ADD(R_0, R_2)$
28: $R_3 \leftarrow MUL(R_2, R_3)$
29: $R_{11} \leftarrow MUL(R_9, R_{11})$
30: $R_0 \leftarrow MUL(R_7, R_{10})$
31: $FP(MUL, ADD) \leftarrow ADD(R_9, R_1)$
32: $R_1 \leftarrow ADD(FP(MUL, ADD), R_{10})$
33: $R_2 \leftarrow MUL(R_3, R_8)$
34: $R_1 \leftarrow MUL(R_3, R_1)$
35: $R_7 \leftarrow MUL(R_0, R_3)$
36: $R_3 \leftarrow MUL(R_0, R_1)$
37: $R_{11} \leftarrow MUL(R_0, R_7)$
38: $R_{12} \leftarrow MUL(R_1, R_7)$
39: $FP(MUL, ADD) \leftarrow MUL(R_4, R_{11})$
40: $R_{12} \leftarrow ADD(FP(MUL, ADD), R_{12}$
41: $FP(ADD, MUL) \leftarrow ADD(R_4, \tilde{U}_{20})$
42: $R_{10} \leftarrow MUL(FP(ADD, MUL), R_{10})$
43: $FP(MUL, ADD) \leftarrow MUL(R_5, R_8)$
44: $R_8 \leftarrow MUL(FP(MUL, ADD), R_7)$
45: $R_7 \leftarrow SQR(R_7)$
46: $Z' \leftarrow MUL(R_7, R_9)$
47: $R_8 \leftarrow MUL(R_2, R_8)$
48: $FP(MUL, ADD) \leftarrow MUL(R_6, R_{11})$
49: $R_6 \leftarrow ADD(FP(MUL, ADD), R_8)$
50: $FP(SQR, MUL) \leftarrow SQR(R_0)$
51: $R_0 \leftarrow MUL(FP(SQR, MUL), R_5)$
52: $FP(ADD, MUL) \leftarrow ADD(R_4, R_5)$
53: $R_0 \leftarrow MUL(FP(ADD, MUL), R_0)$
54: $R_6 \leftarrow ADD(R_0, R_6)$
55: $FP(SQR, ADD) \leftarrow SQR(R_1)$
56: $R_1 \leftarrow ADD(FP(SQR, ADD), R_6)$
57: $U'_0 \leftarrow MUL(R_1, R_9)$
58: $R_4 \leftarrow MUL(R_5, R_{11})$
59: $R_3 \leftarrow MUL(R_3, \tilde{U}_{20})$
60: $R_5 \leftarrow ADD(R_{11}, R_{10})$
61: $R_0 \leftarrow MUL(R_1, R_{12})$
62: $FP(MUL, ADD) \leftarrow MUL(R_7, R_{10})$
63: $R_0 \leftarrow ADD(FP(MUL, ADD), R_0)$
64: $FP(ADD, MUL) \leftarrow ADD(R_1, R_5)$
65: $R_5 \leftarrow MUL(FP(ADD, MUL), R_7)$
66: $FP(MUL, ADD) \leftarrow MUL(R_2, R_{12})$
67: $R_5 \leftarrow ADD(FP(MUL, ADD), R_5)$
68: $FP(MUL, ADD) \leftarrow MUL(R_{11}, \tilde{V}_{20})$
69: $V'_0 \leftarrow ADD(FP(MUL, ADD), R_0)$
70: $FP(ADD, MUL) \leftarrow ADD(R_3, \tilde{V}_{21})$
71: $R_{11} \leftarrow MUL(FP(ADD, MUL), R_{11})$
72: $V'_1 \leftarrow ADD(R_5, R_{11})$
```

Algorithm A.9 The modified register management of mixed addition for *projective* coordinates for an even characteristic when $h_2 = 0$

```
Input: $[U_{11}, U_{10}, V_{11}, V_{10}]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$,
       $h = h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U_1', U_0', V_1', V_0', Z'] = [U_{11}, U_{10}, V_{11}, V_{10}] + [U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$
1: $R_0 \leftarrow U_{11}$, $R_1 \leftarrow U_{10}$,$R_2 \leftarrow V_{11}$, $R_3 \leftarrow V_{10}R_4 \leftarrow Z_2$

2: $FP(MUL, ADD) \leftarrow MUL(R_0, R_4)$
3: $R_5 \leftarrow ADD(FP(MUL, ADD), U_{21})$
4: $FP(MUL, ADD) \leftarrow MUL(R_1, R_4)$
5: $R_6 \leftarrow ADD(FP(MUL, ADD), U_{20})$
6: $FP(MUL, ADD) \leftarrow MUL(R_0, R_5)$
7: $R_7 \leftarrow ADD(FP(MUL, ADD), R_6)$
8: $R_8 \leftarrow MUL(R_6, R_7)$
9: $FP(SQR, MUL) \leftarrow SQR(R_5)$
10: $R_9 \leftarrow MUL(FP(SQR, MUL), R_1)$
11: $R_8 \leftarrow ADD(R_8, R_9)$
12: $FP(MUL, ADD) \leftarrow MUL(R_3, R_4)$
13: $R_3 \leftarrow ADD(FP(MUL, ADD), V_{20})$
14: $FP(MUL, ADD) \leftarrow MUL(R_2, R_4)$
15: $R_2 \leftarrow ADD(FP(MUL, ADD), V_{21})$
16: $R_{10} \leftarrow MUL(R_3, R_7)$
17: $R_{10} \leftarrow MUL(R_2, R_5)$
18: $FP(ADD, MUL) \leftarrow ADD(R_1, R_{10})$
19: $R_1 \leftarrow MUL(FP(ADD, MUL), R_9)$
20: $R_7 \leftarrow ADD(R_5, R_7)$
21: $R_2 \leftarrow ADD(R_2, R_3)$
22: $R_2 \leftarrow MUL(R_0, R_{10})$
23: $FP(MUL, ADD) \leftarrow MUL(R_2, R_7)$
24: $R_2 \leftarrow ADD(FP(MUL, ADD), R_9)$
25: $R_3 \leftarrow MUL(R_8, R_4)$
26: $R_7 \leftarrow MUL(R_1, R_4)$
27: $R_4 \leftarrow MUL(R_2, R_4)$
28: $R_9 \leftarrow MUL(R_3, R_4)$
29: $R_{10} \leftarrow MUL(R_1, R_2)$
30: $R_{11} \leftarrow MUL(R_2, R_4)$
31: $R_1 \leftarrow MUL(R_1, R_4)$
32: $R_{12} \leftarrow MUL(R_{11}, U_{21})$
33: $R_{13} \leftarrow MUL(R_2, R_3)$
34: $R_{14} \leftarrow MUL(R_{10}, R_{13})$
35: $R_1 \leftarrow ADD(R_1, R_{12})$
36: $R_{10} \leftarrow ADD(R_{11}, R_{10})$
37: $FP(ADD, MUL) \leftarrow ADD(R_{13}, R_{14})$
38: $R_{10} \leftarrow MUL(FP(ADD, MUL), R_{10})$
39: $R_{12} \leftarrow ADD(R_{13}, R_{14})$
40: $R_{12} \leftarrow ADD(R_{10}, R_{12})$
41: $R_7 \leftarrow SQR(R_7)$
42: $R_0 \leftarrow MUL(R_0, R_5)$
43: $FP(SQR, MUL) \leftarrow SQR(R_2)$
44: $R_0 \leftarrow MUL(FP(SQR, MUL), R_0)$
45: $R_0 \leftarrow ADD(R_0, R_7)$
46: $R_6 \leftarrow MUL(R_6, R_{11})$
47: $R_0 \leftarrow ADD(R_0, R_6)$
48: $R_2 \leftarrow MUL(R_3, R_8)$
49: $R_8 \leftarrow MUL(R_2, R_5)$
50: $FP(ADD, ADD) \leftarrow ADD(R_8, R_9)$
51: $R_0 \leftarrow ADD(FP(ADD, ADD), R_0)$
52: $R_5 \leftarrow MUL(R_5, R_{11})$
53: $FP(SQR, ADD) \leftarrow SQR(R_3)$
54: $R_3 \leftarrow MUL(FP(SQR, ADD), R_5)$
55: $R_0 \leftarrow MUL(R_0, R_9)$
56: $R_4 \leftarrow SQR(R_4)$
57: $FP(MUL, ADD) \leftarrow MUL(R_{13}, V_{21})$
58: $R_7 \leftarrow ADD(FP(MUL, ADD), R_3)$
59: $FP(ADD, MUL) \leftarrow ADD(R_7, R_{12})$
60: $R_7 \leftarrow MUL(FP(ADD, MUL), R_4)$
61: $FP(ADD, MUL) \leftarrow ADD(R_1, R_3)$
62: $R_2 \leftarrow MUL(FP(ADD, MUL), R_3)$
63: $V_1' \leftarrow ADD(R_2, R_7)$
64: $FP(MUL, ADD) \leftarrow MUL(R_{13}, V_{20})$
65: $R_7 \leftarrow ADD(FP(MUL, ADD), R_{14})$
66: $R_7 \leftarrow MUL(R_4, R_7)$
67: $FP(ADD, MUL) \leftarrow ADD(R_1, R_3)$
68: $R_0 \leftarrow MUL(FP(ADD, MUL), R_0)$
69: $R_1 \leftarrow ADD(R_0, R_7)$
```

Algorithm A.10 The modified register management of divisor doubling for *projective* coordinates for an even characteristic when $h_2 \neq 0$

Input: $[U_1, U_0, V_1, V_0, Z]$,$h = h_2x^2 + h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U_1', U_0', V_1', V_0', Z'] = 2[U_1, U_0, V_1, V_0, Z]$

```
1: $R_0 \leftarrow V_1,\ R_1 \leftarrow V_0, R_2 \leftarrow U_1,\ R_3 \leftarrow U_0, R_4 \leftarrow Z$
2: $R_5 \leftarrow cMUL\,(R_4,\ h_1)$
3: $R_6 \leftarrow cMUL\,(R_4,\ h_0)$
4: $FP(MUL,\ ADD) \leftarrow MUL\,(R_5,\ U_1)$
5: $R_8 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_2)$
6: $FP(MUL,\ ADD) \leftarrow MUL\,(R_5,\ U_0)$
7: $R_5 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_3)$
8: $R_9 \leftarrow SQR\,(R_8)$
9: $R_9 \leftarrow SQR\,(R_5)$
10: $FP(ADD,\ MUL) \leftarrow MUL\,(R_8)$
11: $R_{10} \leftarrow MUL\,(FP(ADD,\ MUL),\ U_1)$
12: $FP(MUL,\ ADD) \leftarrow MUL\,(R_5,\ R_9)$
13: $R_{10} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{10})$
14: $R_6 \leftarrow MUL\,(R_6,\ R_{10})$
15: $Z' \leftarrow MUL\,(R_6,\ R_4)$
16: $FP(MUL,\ ADD) \leftarrow MUL\,(R_7,\ V_0)$
17: $R_{11} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_1)$
18: $FP(MUL,\ ADD) \leftarrow MUL\,(R_7,\ V_1)$
19: $R_7 \leftarrow ADD\,(FP(MUL,\ ADD),\ V_0)$
20: $R_{13} \leftarrow MUL\,(R_7,\ R_8)$
21: $R_{15} \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{13})$
22: $R_{14} \leftarrow ADD\,(R_8,\ R_9)$
23: $FP(ADD,\ MUL) \leftarrow ADD\,(R_7,\ R_{11})$
24: $R_{14} \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{14})$
25: $FP(SUB,\ SUB) \leftarrow SUB\,(R_{12},\ R_{14})$
26: $R_{14} \leftarrow SUB\,(FP(SUB,\ SUB),\ R_{15})$
27: $R_{14} \leftarrow MUL\,(R_{14},\ R_{10})$
28: $FP(MUL,\ ADD) \leftarrow MUL\,(U_0\, R_{13})$
29: $R_9 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{12})$
30: $R_{11} \leftarrow MUL\,(R_7,\ R_4)$
31: $R_{12} \leftarrow MUL\,(R_7,\ R_{10})$
32: $R_{10} \leftarrow MUL\,(R_4,\ R_9)$
33: $R_{13} \leftarrow MUL\,(R_{10},\ R_{11})$
34: $R_{10} \leftarrow SQR\,(R_{10})$
35: $R_9 \leftarrow MUL\,(R_7,\ R_9)$
36: $R_7 \leftarrow SQR\,(R_7)$
37: $R_9 \leftarrow MUL\,(R_{10},\ R_{14})$
38: $R_{14} \leftarrow SQR\,(R_{11})$
39: $R_{11} \leftarrow MUL\,(Z',\ R_{11})$
40: $R_4 \leftarrow MUL\,(R_4,\ R_7)$
41: $R_{15} \leftarrow ADD\,(R_7,\ R_9)$
42: $R_{13} \leftarrow MUL\,(R_{12},\ R_4)$
43: $R_9 \leftarrow MUL\,(R_9,\ R_3)$
44: $FP(ADD,\ MUL) \leftarrow ADD\,(R_2,\ R_3)$
45: $R_2 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{15})$
46: $FP(SUB,\ SUB) \leftarrow SUB\,(R_2,\ R_9)$
47: $R_2 \leftarrow SUB\,(FP(SUB,\ SUB),\ R_4)$
48: $R_4 \leftarrow ADD\,(R_4,\ R_{13})$
49: $R_3 \leftarrow MUL\,(U_1,\ R_7)$
50: $FP(MUL,\ ADD) \leftarrow ADD\,(R_6,\ R_3)$
51: $R_6 \leftarrow MUL\,(FP(MUL,\ ADD),\ R_8)$
52: $R_3 \leftarrow ADD\,(R_{10},\ R_6)$
53: $R_5 \leftarrow MUL\,(R_5,\ R_7)$
54: $R_5 \leftarrow ADD\,(R_5,\ R_3)$
55: $R_3 \leftarrow ADD\,(R_5,\ R_6)$
56: $FP(MUL,\ ADD) \leftarrow MUL\,(R_7,\ R_8)$
57: $R_4 \leftarrow ADD\,(R_4,\ R_5)$
58: $R_6 \leftarrow MUL\,(R_3,\ R_4)$
59: $R_7 \leftarrow MUL\,(R_4,\ R_5)$
60: $FP(MUL,\ ADD) \leftarrow MUL\,(R_0,\ R_{12})$
61: $R_2 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_2)$
62: $R_4 \leftarrow ADD\,(R_4,\ R_2)$
63: $FP(ADD,\ MUL) \leftarrow ADD\,(R_3,\ R_4)$
64: $R_2 \leftarrow ADD\,(FP(ADD, MUL),\ R_{14})$
65: $R_0 \leftarrow ADD\,(R_2,\ R_6)$
66: $FP(MUL,\ ADD) \leftarrow MUL\,(R_1,\ R_{12})$
67: $R_4 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_9)$
68: $FP(ADD,\ MUL) \leftarrow ADD\,(R_2,\ R_4)$
69: $R_4 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{14})$
70: $R_1 \leftarrow ADD\,(R_6,\ R_4)$
```

Algorithm A.11 The modified register management of divisor addition for *projective* coordinates for an even characteristic when $h_2 \neq 0$

```
Input: $[U_{11}, U_{10}, V_{11}, V_{10}, Z_1]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$,
       $h = h_2x^2 + h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U_1', U_0', V_1', V_0', Z'] = [U_{11}, U_{10}, V_{11}, V_{10}, Z_1] + [U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$
 1: $R_0 \leftarrow U_{21},\ R_1 \leftarrow U_{20}$,$R_2 \leftarrow V_{21},\ R_3 \leftarrow V_{20}$,$R_4 \leftarrow Z_1,\ R_5 \leftarrow Z_2$

 2: $R_6 \leftarrow MUL\,(R_4,\ R_5)$
 3: $R_7 \leftarrow MUL\,(R_6,\ U_{21})$
 4: $R_6 \leftarrow MUL\,(R_6,\ U_{20})$
 5: $R_{11} \leftarrow MUL\,(R_{10},\ U_{11})$
 6: $R_{12} \leftarrow MUL\,(R_{11},\ R_6)$
 7: $FP(MUL,\ ADD) \leftarrow MUL\,(R_9,\ V_{10})$
 8: $R_{13} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_3)$
 9: $FP(MUL,\ ADD) \leftarrow MUL\,(R_9,\ V_{11})$
10: $R_9 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_2)$
11: $R_0 \leftarrow SQR\,(R_{13})$
12: $FP(MUL,\ ADD) \leftarrow ADD\,(R_{11},\ R_{10})$
13: $R_{11} \leftarrow MUL\,(R_{11},\ FP(MUL,\ ADD))$
14: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{10},\ R_9)$
15: $R_9 \leftarrow ADD\,(R_9,\ FP(MUL,\ ADD))$
16: $R_9 \leftarrow MUL\,(R_9,\ R_{14})$
17: $FP(SQR,\ MUL) \leftarrow SQR\,(R_9)$
18: $R_{13} \leftarrow MUL\,(FP(SQR,\ MUL),\ R_{10})$
19: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{13},\ U_{10})$
20: $R_{12} \leftarrow ADD\,(R_{12},\ FP(MUL,\ ADD))$
21: $R_8 \leftarrow MUL\,(R_8,\ R_{12})$
22: $R_{13} \leftarrow MUL\,(R_8,\ R_8)$
23: $R_8 \leftarrow MUL\,(R_8,\ R_4)$
24: $R_{14} \leftarrow ADD\,(1,\ U_{11})$
25: $R_{14} \leftarrow MUL\,(R_{14},\ R_{10})$
26: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{10},\ U_{10})$
27: $R_{10} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{11})$
28: $FP(SQR,\ MUL) \leftarrow SQR\,(R_{13})$
29: $R_{11} \leftarrow MUL\,(R_{12},\ FP(SQR,\ MUL))$
30: $R_{12} \leftarrow MUL\,(R_{10},\ R_4)$
31: $R_{10} \leftarrow MUL\,(R_{10},\ R_9)$
32: $R_6 \leftarrow MUL\,(R_6,\ R_{14})$
33: $R_9 \leftarrow MUL\,(R_{10},\ R_{14})$
34: $FP(MUL,\ ADD) \leftarrow MUL\,(R_2,\ R_8)$
35: $R_9 \leftarrow ADD\,(R_9,\ FP(MUL,\ ADD))$
36: $R_{16} \leftarrow MUL\,(R_8,\ R_4)$
37: $R_{17} \leftarrow MUL\,(R_{14},\ U_{11})$
38: $R_{13} \leftarrow ADD\,(R_{13},\ R_{17})$
39: $R_{10} \leftarrow MUL\,(R_{10},\ R_{13})$
40: $R_{18} \leftarrow ADD\,(R_0,\ R_1)$
41: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{12},\ R_{13})$
42: $R_{10} \leftarrow ADD\,(R_{10},\ FP(MUL,\ ADD))$
43: $R_6 \leftarrow ADD\,(R_6,\ R_{10})$
44: $R_3 \leftarrow MUL\,(R_3,\ R_{11})$
45: $R_2 \leftarrow MUL\,(R_2,\ R_{11})$
46: $R_0 \leftarrow MUL\,(R_0,\ R_{14})$
47: $FP(ADD,\ MUL) \leftarrow ADD\,(R_{10},\ R_{14})$
48: $R_{10} \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{18})$
49: $R_1 \leftarrow MUL\,(R_1,\ R_{10})$
50: $R_{10} \leftarrow ADD\,(R_{10},\ R_1)$
51: $R_1 \leftarrow ADD\,(R_1,\ R_3)$
52: $FP(SQR,\ ADD) \leftarrow SQR\,(R_{10})$
53: $R_2 \leftarrow ADD\,(FP(SQR,\ ADD),\ R_3)$
54: $R_0 \leftarrow ADD\,(R_0,\ R_{13})$
55: $R_0 \leftarrow ADD\,(R_0,\ R_9)$
56: $R_3 \leftarrow MUL\,(R_9,\ R_0)$
57: $R_{10} \leftarrow MUL\,(R_{16},\ h_1)$
58: $FP(ADD,\ MUL) \leftarrow ADD\,(R_6,\ R_{10})$
59: $R_0 \leftarrow MUL\,(R_0,\ FP(ADD,\ MUL))$
60: $R_{12} \leftarrow ADD\,(R_{12},\ R_{13})$
61: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{16},\ h_1)$
62: $R_2 \leftarrow ADD\,(R_2,\ FP(MUL,\ ADD))$
63: $FP(ADD,\ MUL) \leftarrow ADD\,(R_2,\ R_6)$
64: $R_2 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{17})$
65: $R_2 \leftarrow ADD\,(R_2,\ R_3)$
66: $R_3 \leftarrow MUL\,(R_{16},\ h_0)$
67: $FP(ADD,\ MUL) \leftarrow ADD\,(R_1,\ R_3)$
68: $R_1 \leftarrow MUL\,(R_1,\ FP(ADD,\ MUL))$
69: $R_3 \leftarrow MUL\,(R_{16}, R_{17})$
70: $R_{11} \leftarrow MUL\,(R_7,\ h_2)$
```

Algorithm A.12 The modified register management of mixed Addition for projective coordinates for an even characteristic when $h_2 \neq 0$

```
Input: $[U_{11}, U_{10}, V_{11}, V_{10}]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$,
       $h = h_2x^2 + h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U'_1, U'_0, V'_1, V'_0, Z']$
 1: $R_0 \leftarrow V_{21}$, $R_1 \leftarrow V_{20}$,$R_2 \leftarrow U_{21}$, $R_3 \leftarrow U_{20}$,$R_4 \leftarrow Z_2$

 2: $FP(MUL, ADD) \leftarrow MUL\,(R_5, U_{11})$
 3: $R_8 \leftarrow ADD\,(FP(MUL, ADD), R_2)$
 4: $FP(MUL, ADD) \leftarrow MUL\,(R_5, U_{10})$
 5: $R_5 \leftarrow ADD\,(FP(MUL, ADD), R_3)$
 6: $FP(MUL, ADD) \leftarrow MUL\,(R_8, U_{11})$
 7: $R_9 \leftarrow ADD\,(FP(MUL, ADD), R_5)$
 8: $FP(SQR, MUL) \leftarrow SQR\,(R_8)$
 9: $R_{10} \leftarrow MUL\,(FP(SQR, MUL), U_{10})$
10: $FP(MUL, ADD) \leftarrow MUL\,(R_5, R_9)$
11: $R_{10} \leftarrow ADD\,(FP(MUL, ADD), R_{10})$
12: $R_6 \leftarrow MUL\,(R_6, R_{10})$
13: $Z' \leftarrow MUL\,(R_6, R_4)$
14: $R_6 \leftarrow SQR\,(R_6)$
15: $FP(MUL, ADD) \leftarrow MUL\,(R_7, V_{10})$
16: $R_{11} \leftarrow ADD\,(FP(MUL, ADD), R_1)$
17: $FP(MUL, ADD) \leftarrow MUL\,(R_7, V_{11})$
18: $R_7 \leftarrow ADD\,(FP(MUL, ADD), V_{21})$
19: $R_{13} \leftarrow MUL\,(R_7, R_8)$
20: $FP(ADD, MUL) \leftarrow ADD\,(U_{11}, 1)$
21: $R_{15} \leftarrow MUL\,(FP(ADD, MUL), R_{13})$
22: $R_{14} \leftarrow ADD\,(R_8, R_9)$
23: $FP(ADD, MUL) \leftarrow ADD\,(R_7, R_{11})$
24: $R_{14} \leftarrow MUL\,(FP(ADD, MUL), R_{14})$
25: $FP(SUB, SUB) \leftarrow SUB\,(R_{12}, R_{14})$
26: $R_{14} \leftarrow SUB\,(FP(SUB, SUB), R_{15})$
27: $R_{14} \leftarrow MUL\,(R_{14}, R_{10})$
28: $FP(MUL, ADD) \leftarrow MUL\,(U_{10}\, R_{13})$
29: $R_9 \leftarrow ADD\,(FP(MUL, ADD), R_{12})$
30: $R_{11} \leftarrow MUL\,(R_7, R_4)$
31: $R_{12} \leftarrow MUL\,(R_7, R_{10})$
32: $R_{10} \leftarrow MUL\,(R_4, R_9)$
33: $R_{13} \leftarrow MUL\,(R_{10}, R_{11})$
34: $R_{10} \leftarrow SQR\,(R_{10})$
35: $R_9 \leftarrow MUL\,(R_7, R_9)$
36: $R_7 \leftarrow SQR\,(R_7)$
37: $R_9 \leftarrow MUL\,(R_{10}, R_{14})$
38: $R_{14} \leftarrow SQR\,(R_{11})$
39: $z'_2 \leftarrow SQR\,(Z'_2)$
40: $R_{11} \leftarrow MUL\,(Z'_2, R_{11})$
41: $z'_4 \leftarrow MUL\,(R_{14}, R_{11})$
42: $R_4 \leftarrow MUL\,(R_4, R_7)$
43: $R_{15} \leftarrow ADD\,(R_7, R_9)$
44: $R_{13} \leftarrow MUL\,(R_{12}, R_4)$
45: $R_9 \leftarrow MUL\,(R_9, R_3)$
46: $FP(ADD, MUL) \leftarrow ADD\,(R_2, R_3)$
47: $R_2 \leftarrow MUL\,(FP(ADD, MUL), R_{15})$
48: $FP(SUB, SUB) \leftarrow SUB\,(R_2, R_9)$
49: $R_2 \leftarrow SUB\,(FP(SUB, SUB), R_4)$
50: $R_4 \leftarrow ADD\,(R_4, R_{13})$
51: $R_3 \leftarrow MUL\,(U_{11}, R_7)$
52: $FP(MUL, ADD) \leftarrow ADD\,(R_6, R_3)$
53: $R_6 \leftarrow MUL\,(FP(MUL, ADD), R_8)$
54: $R_3 \leftarrow ADD\,(R_{10}, R_6)$
55: $R_5 \leftarrow MUL\,(R_5, R_7)$
56: $R_6 \leftarrow cMUL\,(h_1, R_{11})$
57: $R_5 \leftarrow ADD\,(R_5, R_3)$
58: $R_3 \leftarrow ADD\,(R_5, R_6)$
59: $FP(MUL, ADD) \leftarrow MUL\,(R_7, R_8)$
60: $R_5 \leftarrow ADD\,(FP(MUL, ADD), z'_2)$
61: $R_4 \leftarrow ADD\,(R_4, R_5)$
62: $R_6 \leftarrow MUL\,(R_3, R_4)$
63: $R_7 \leftarrow MUL\,(R_4, R_5)$
64: $FP(MUL, ADD) \leftarrow MUL\,(R_0, R_{12})$
65: $R_2 \leftarrow ADD\,(FP(MUL, ADD), R_2)$
66: $R_4 \leftarrow ADD\,(R_4, R_2)$
67: $FP(ADD, MUL) \leftarrow ADD\,(R_3, R_4)$
68: $R_2 \leftarrow ADD\,(FP(ADD,MUL), R_{14})$
69: $R_0 \leftarrow ADD\,(R_2, R_6)$
70: $FP(MUL, ADD) \leftarrow MUL\,(R_1, R_{12})$
71: $R_4 \leftarrow ADD\,(FP(MUL, ADD), R_9)$
72: $FP(ADD, MUL) \leftarrow ADD\,(R_2, R_4)$
73: $R_4 \leftarrow MUL\,(FP(ADD, MUL), R_{14})$
```

A.3 Recent Coordinates ($\mathcal{R}$)

The classification of the different types of genus 2 curves in a characteristic 2 allows significant increase in the speed of the formulas for doubling which are included in this section for recent coordinates. These coordinates have the advantage of allowing efficient doubling explicit formulas but the additions are more expensive. However, the doubling operation is usually the operation that is repeated in each loop of the performance of the divisor multiplication operation. The formulas for recent coordinates can be found in Algorithms 5.4, A.13, A.14, A.15, A.16, and A.17.

.

Algorithm A.13 The modified register management of divisor addition for *recent* coordinates for an even characteristic when $h_2 = 0$

```
Input: $[U_{11}, U_{10}, V_{11}, V_{10}, Z_1, z_1][U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$,
       $h = h_1x + h_0, f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U_1', U_0', V_1', V_0', Z', z'] = [U_{11}, U_{10}, V_{11}, V_{10}, Z_1, z_1] + [U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$
1: $R_0 \leftarrow U_{21}, R_1 \leftarrow U_{11}, R_2 \leftarrow U_{10}, R_3 \leftarrow V_{11}, R_4 \leftarrow V_{10}, R_5 \leftarrow Z_1, R_6 \leftarrow Z_2, R_7 \leftarrow z_1$

2: $R_9 \leftarrow MUL(R_5, R_6)$
3: $R_{10} \leftarrow SQR(R_9)$
4: $R_{11} \leftarrow MUL(R_0, R_5)$
5: $R_{12} \leftarrow MUL(R_5, U_{20})$
6: $R_3 \leftarrow MUL(R_7, V_{21})$
7: $R_{13} \leftarrow MUL(R_{12}, V_{20})$
8: $R_0 \leftarrow ADD(R_2, R_{12})$
9: $FP(MUL, ADD) \leftarrow MUL(R_{13}, V_{10})$
10: $R_{13} \leftarrow ADD(R_9, FP(MUL, ADD))$
11: $R_{19} \leftarrow ADD(R_{13}, R_2)$
12: $FP(MUL, ADD) \leftarrow MUL(R_{11}, R_7)$
13: $R_1 \leftarrow ADD(R_1, R_{11})$
14: $R_4 \leftarrow MUL(R_2, R_7)$
15: $FP(MUL, ADD) \leftarrow MUL(R_0, R_7)$
16: $R_2 \leftarrow ADD(FP(MUL, ADD), R_5)$
17: $R_2 \leftarrow MUL(R_2, R_0)$
18: $R_3 \leftarrow MUL(R_0, R_2)$
19: $R_{17} \leftarrow MUL(R_2, R_{12})$
20: $FP(MUL, ADD) \leftarrow MUL(R_1, R_2)$
21: $R_2 \leftarrow ADD(FP(MUL, ADD), R_4)$
22: $FP(MUL, ADD) \leftarrow MUL(R_4, R_{13})$
23: $R_0 \leftarrow ADD(FP(MUL, ADD), R_2)$
24: $FP(MUL, ADD) \leftarrow MUL(R_2, R_7)$
25: $R_6 \leftarrow ADD(R_4, FP(MUL, ADD))$
26: $R_9 \leftarrow MUL(R_6, R_9)$
27: $R_{13} \leftarrow ADD(R_{19}, R_{13})$
28: $FP(ADD, MUL) \leftarrow ADD(R_{13}, R_3)$
29: $R_9 \leftarrow MUL(FP(ADD, MUL), R_4)$
30: $R_3 \leftarrow ADD(R_2, R_5)$
31: $R_8 \leftarrow MUL(R_9, R_8)$
32: $R_4 \leftarrow MUL(R_2, R_{12})$
33: $FP(MUL, ADD) \leftarrow MUL(R_{24}, R_2)$
34: $R_9 \leftarrow ADD(R_9, FP(MUL, ADD))$
35: $R_{16} \leftarrow MUL(R_{16}, R_7)$
36: $R_{12} \leftarrow MUL(R_9, R_{12})$
37: $R_1 \leftarrow MUL(R_9, R_{16})$
38: $FP(ADD, MUL) \leftarrow ADD(R_{19}, R_{23})$
39: $R_{19} \leftarrow MUL(FP(ADD, MUL), R_{22})$
40: $R_{12} \leftarrow MUL(R_{12}, R_{13})$
41: $R_{16} \leftarrow MUL(R_{16}, R_{10})$
42: $R_{10} \leftarrow MUL(R_{10}, R_{14})$
43: $FP(MUL, ADD) \leftarrow MUL(R_{14}, R_{16})$
44: $R_{16} \leftarrow ADD(FP(MUL, ADD), R_{13})$
45: $R_{13} \leftarrow MUL(R_{13}, R_{15})$
46: $R_{10} \leftarrow MUL(R_{10}, R_{11})$
47: $R_{11} \leftarrow ADD(R_{11}, R_{15})$
48: $FP(MUL, ADD) \leftarrow MUL(R_{11}, R_{16})$
49: $R_{11} \leftarrow ADD(FP(MUL, ADD), R_{13})$
50: $R_{13} \leftarrow ADD(R_{13}, R_3)$
51: $R_{11} \leftarrow ADD(R_{11}, R_{10})$
52: $R_{11} \leftarrow ADD(R_{11}, R_{12})$
53: $FP(ADD, MUL) \leftarrow ADD(R_{10}, R_2)$
54: $R_{12} \leftarrow MUL(FP(ADD, MUL), R_{16})$
55: $R_{12} \leftarrow ADD(R_{12}, R_9)$
56: $FP(ADD, MUL) \leftarrow ADD(R_{12}, R_1)$
57: $R_{16} \leftarrow MUL(R_{14}, FP(ADD, MUL))$
58: $R_{10} \leftarrow MUL(R_{16}, R_{10})$
59: $R_{15} \leftarrow MUL(R_4, R_8)$
60: $R_2 \leftarrow MUL(R_8, R_1)$
61: $R_{11} \leftarrow ADD(R_{12}, R_{10})$
62: $FP(ADD, MUL) \leftarrow ADD(R_{13}, R_{11})$
63: $R_{12} \leftarrow MUL(FP(ADD, MUL), R_2)$
64: $FP(ADD, MUL) \leftarrow ADD(R_{12}, R_{13})$
65: $R_3 \leftarrow MUL(FP(ADD, MUL), R_{12})$
66: $R_{10} \leftarrow ADD(R_{10}, R_{13})$
67: $FP(ADD, MUL) \leftarrow ADD(R_{11}, R_{12})$
68: $R_{11} \leftarrow MUL(FP(ADD, MUL), R_{16})$
69: $R_{13} \leftarrow MUL(R_{16}, R_9)$
70: $R_{11} \leftarrow SQR(R_{11})$
71: $R_{12} \leftarrow SQR(R_{12})$
```

Algorithm A.14 The modified register management of mixed addition for *recent* coordinates for an even characteristic when $h_2 = 0$

Input: $[U_{11}, U_{10}, V_{11}, V_{10}]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$,
$h = h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.

Output: $[U_1', U_0', V_1', V_0', Z', z'] = [U_{11}, U_{10}, V_{11}, V_{10}] + [U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$

```
 1: $R_0 \leftarrow U_{21},\ R_1 \leftarrow U_{20}, R_2 \leftarrow V_{21},\ R_3 \leftarrow V_{20}, R_4 \leftarrow Z_2,\ R_5 \leftarrow z_2$

 2: $FP(MUL,\ ADD) \leftarrow MUL\,(R_4,\ U_{11})$
 3: $R_{10} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_0)$
 4: $FP(MUL,\ ADD) \leftarrow MUL\,(R_6,\ U_{10})$
 5: $R_6 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_1)$
 6: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{10},\ U_{11})$
 7: $R_{11} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_6)$
 8: $R_{12} \leftarrow MUL\,(R_{11},\ R_6)$
 9: $FP(MUL,\ ADD) \leftarrow MUL\,(R_9,\ V_{10})$
10: $R_{13} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_3)$
11: $FP(MUL,\ ADD) \leftarrow MUL\,(R_9,\ V_{11})$
12: $R_9 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_2)$
13: $R_4 \leftarrow ADD\,(R_{11},\ R_{10})$
14: $R_{11} \leftarrow MUL\,(R_{11},\ R_{13})$
15: $R_{10} \leftarrow MUL\,(R_{10},\ R_9)$
16: $R_9 \leftarrow ADD\,(R_9,\ R_{13})$
17: $R_9 \leftarrow MUL\,(R_9,\ R_{14})$
18: $R_{13} \leftarrow MUL\,(R_{10},\ R_{10})$
19: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{13},\ U_{10})$
20: $R_{12} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{13})$
21: $R_8 \leftarrow MUL\,(R_8,\ R_{12})$
22: $R_8 \leftarrow MUL\,(R_8,\ R_4)$
23: $R_4 \leftarrow MUL\,(R_1,\ R_{10})$
24: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{10},\ U_{10})$
25: $R_{10} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{11})$
26: $FP(SQR,\ MUL) \leftarrow SQR\,(R_9,\ R_{14})$
27: $R_{11} \leftarrow MUL\,(R_{12},\ FP(SQR,\ MUL))$
28: $R_{12} \leftarrow MUL\,(R_{10},\ R_4)$
29: $R_{10} \leftarrow MUL\,(R_{10},\ R_9)$
30: $R_4 \leftarrow MUL\,(R_9,\ R_9)$
31: $R_4 \leftarrow MUL\,(R_4,\ R_9)$
32: $R_6 \leftarrow MUL\,(R_6,\ R_4)$
33: $R_9 \leftarrow ADD\,(R_9,\ R_5)$
34: $R_6 \leftarrow MUL\,(R_8,\ R_4)$
35: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{14},\ U_{11})$
36: $R_{13} \leftarrow ADD\,(R_{13},\ FP(MUL,\ ADD))$
37: $R_{10} \leftarrow MUL\,(R_{10},\ R_{13})$
38: $R_{13} \leftarrow MUL\,(R_{12},\ R_4)$
39: $R_7 \leftarrow MUL\,(R_8,\ R_4)$
40: $R_8 \leftarrow ADD\,(R_0,\ R_1)$
41: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{12},\ R_{12})$
42: $R_{10} \leftarrow ADD\,(R_{10},\ FP(MUL,\ ADD))$
43: $R_6 \leftarrow ADD\,(R_6,\ R_{10})$
44: $R_3 \leftarrow MUL\,(R_3,\ R_{11})$
45: $R_2 \leftarrow MUL\,(R_2,\ R_{11})$
46: $FP(MUL,\ ADD) \leftarrow MUL\,(R_0,\ R_{14})$
47: $R_{10} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{14})$
48: $FP(MUL,\ ADD) \leftarrow MUL\,(R_1,\ R_{10})$
49: $R_1 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_3)$
50: $R_2 \leftarrow ADD\,(R_2,\ R_3)$
51: $R_0 \leftarrow ADD\,(R_0,\ R_{13})$
52: $FP(ADD,\ MUL) \leftarrow ADD\,(R_0,\ R_9)$
53: $R_3 \leftarrow MUL\,(R_9,\ FP(ADD,\ MUL))$
54: $R_{10} \leftarrow MUL\,(R_{16},\ h_1)$
55: $FP(ADD,\ MUL) \leftarrow ADD\,(R_6,\ R_{10})$
56: $R_0 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_6)$
57: $R_{12} \leftarrow ADD\,(R_{12},\ R_{13})$
58: $R_2 \leftarrow ADD\,(R_2,\ R_{10})$
59: $FP(ADD,\ MUL) \leftarrow ADD\,(R_2,\ R_6)$
60: $R_2 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{13})$
61: $R_2 \leftarrow ADD\,(R_2,\ R_3)$
62: $FP(ADD,\ MUL) \leftarrow ADD\,(R_1,\ R_3)$
63: $R_1 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_{13})$
64: $R_3 \leftarrow MUL\,(R_{16}, R_{13})$
65: $R_{11} \leftarrow SQR\,(R_7)$
66: $R_0 \leftarrow SQR\,(R_0)$
```

Algorithm A.15 The modified register management of divisor doubling for *recent* coordinates for an even characteristic when $h_2 \neq 0$

Input: $[U_1, U_0, V_1, V_0, Z, z]$, $h = h_2x^2 + h_1x + h_0$, $f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U_1', U_0', V_1', V_0', Z', z'] = 2[U_1, U_0, V_1, V_0, Z, z]$.

```
1: $R_0 \leftarrow U_1,\ R_1 \leftarrow U_0, R_2 \leftarrow V_1,\ R_3 \leftarrow V_0, R_4 \leftarrow Z,\ R_5 \leftarrow z$

2: $R_{10} \leftarrow MUL\,(R_4,\ h_1)$
3: $R_{11} \leftarrow MUL\,(R_4,\ h_0)$
4: $FP(MUL,\ ADD) \leftarrow MUL\,(h_2,\ R_1)$
5: $R_{13} \leftarrow ADD\,(R_{13},\ FP(MUL,\ ADD))$
6: $R_{16} \leftarrow MUL\,(R_6,\ h_0)$
7: $FP(ADD,\ MUL) \leftarrow ADD\,(R_{13},\ R_6)$
8: $R_{13} \leftarrow MUL\,(R_6,\ FP(ADD,\ MUL))$
9: $R_{13} \leftarrow MUL\,(R_9,\ R_{13})$
10: $FP(MUL,\ ADD) \leftarrow MUL\,(R_0,\ h_2)$
11: $R_{10} \leftarrow ADD\,(R_{10},\ FP(MUL,\ ADD))$
12: $R_6 \leftarrow MUL\,(R_0,\ R_{10})$
13: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{12},\ R_1)$
14: $R_{11} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{12})$
15: $R_{11} \leftarrow MUL\,(R_{11},\ R_1)$
16: $FP(MUL,\ ADD) \leftarrow MUL\,(R_6,\ h_2)$
17: $R_{13} \leftarrow ADD\,(R_{13},\ FP(MUL,\ ADD))$
18: $R_7 \leftarrow MUL\,(R_{12},\ R_7)$
19: $R_{15} \leftarrow MUL\,(R_2,\ h_2)$
20: $FP(MUL,\ ADD) \leftarrow MUL\,(R_5,\ R_{13})$
21: $R_{11} \leftarrow ADD\,(R_{11},\ FP(MUL,\ ADD))$
22: $R_8 \leftarrow MUL\,(R_8,\ R_{11})$
23: $R_9 \leftarrow MUL\,(R_8,\ R_9)$
24: $R_{11} \leftarrow MUL\,(R_2,\ R_2)$
25: $FP(MUL,\ ADD) \leftarrow MUL\,(R_0,\ R_7)$
26: $R_{11} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{11})$
27: $R_{11} \leftarrow ADD\,(R_{11},\ R_4)$
28: $R_{13} \leftarrow ADD\,(R_{12},\ R_{10})$
29: $R_{12} \leftarrow MUL\,(R_{11},\ R_{12})$
30: $R_{10} \leftarrow MUL\,(R_7,\ R_{10})$
31: $FP(ADD,\ MUL) \leftarrow ADD\,(R_{11},\ R_7)$
32: $R_7 \leftarrow MUL\,(R_{13},\ FP(ADD,\ MUL))$
33: $R_7 \leftarrow ADD\,(R_7,\ R_{12})$
34: $R_{13} \leftarrow MUL\,(R_{10},\ R_{13})$
35: $FP(ADD,\ MUL) \leftarrow ADD\,(R_{12},\ R_{10})$
36: $R_{12} \leftarrow MUL\,(FP(ADD,\ MUL),\ h_2)$
37: $FP(MUL,\ ADD) \leftarrow MUL\,(R_6,\ h_1)$
38: $R_{11} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{13})$
39: $FP(MUL,\ ADD) \leftarrow MUL\,(R_7,\ R_{11})$
40: $R_{11} \leftarrow ADD\,(R_{12},\ FP(MUL,\ ADD))$
41: $R_{11} \leftarrow MUL\,(R_9,\ R_{11})$
42: $R_6 \leftarrow MUL\,(R_7,\ R_6)$
43: $R_3 \leftarrow MUL\,(R_8,\ R_3)$
44: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{10},\ R_{10})$
45: $R_8 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{11})$
46: $R_{11} \leftarrow MUL\,(R_{10},\ R_6)$
47: $R_{10} \leftarrow MUL\,(R_{10},\ R_7)$
48: $FP(MUL,\ ADD) \leftarrow MUL\,(R_6,\ R_7)$
49: $R_{12} \leftarrow ADD\,(R_7,\ FP(MUL,\ ADD))$
50: $R_7 \leftarrow MUL\,(R_7,\ R_0)$
51: $R_{10} \leftarrow MUL\,(R_{10},\ R_1)$
52: $R_{11} \leftarrow ADD\,(R_7,\ R_{11})$
53: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{12},\ R_0)$
54: $R_0 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{10})$
55: $R_0 \leftarrow ADD\,(R_0,\ R_2)$
56: $FP(ADD,\ MUL) \leftarrow ADD\,(R_0,\ R_8)$
57: $R_1 \leftarrow MUL\,(FP(ADD,\ MUL),\ R_6)$
58: $R_2 \leftarrow MUL\,(R_1,\ R_3)$
59: $R_3 \leftarrow MUL\,(R_3,\ R_9)$
60: $R_{12} \leftarrow MUL\,(R_3,\ h_2)$
61: $FP(MUL,\ ADD) \leftarrow MUL\,(R_3,\ h_2)$
62: $R_{13} \leftarrow ADD\,(R_7,\ FP(MUL,\ ADD))$
63: $R_{11} \leftarrow ADD\,(R_{11},\ R_{12})$
64: $R_{11} \leftarrow ADD\,(R_{11},\ R_{13})$
65: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{11},\ R_8)$
66: $R_2 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_2)$
67: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{11},\ R_{13})$
68: $R_0 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_0)$
69: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{10},\ h_1)$
70: $R_0 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{11})$
71: $R_{11} \leftarrow MUL\,(R_{10},\ h_0)$
72: $R_2 \leftarrow ADD\,(R_2,\ R_{11})$
```

Algorithm A.16 The modified register management of divisor addition for *recent* coordinates for an even characteristic when $h_2 \neq 0$

```
Input: $[U_{11}, U_{10}, V_{11}, V_{10}, Z_1, z_1]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$,
       $h = h_2x^2 + h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U_1', U_0', V_1', V_0', Z', z'] = [U_{11}, U_{10}, V_{11}, V_{10}, Z_1, z_1] + [U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$
1: $R_0 \leftarrow U_{11}, R_1 \leftarrow U_{10}, R_2 \leftarrow V_{11}, R_3 \leftarrow V_{10}, R_4 \leftarrow Z_1, R_5 \leftarrow Z_2$

2: $R_6 \leftarrow MUL(R_4, U_{21})$
3: $R_9 \leftarrow MUL(R_4, U_{20})$
4: $R_{10} \leftarrow MUL(R_1, V_{21})$
5: $R_{12} \leftarrow MUL(R_0, V_{20})$
6: $R_{13} \leftarrow ADD(R_4, R_5)$
7: $R_{12} \leftarrow MUL(R_{12}, R_9)$
8: $FP(MUL, ADD) \leftarrow MUL(R_{13}, R_9)$
9: $R_{13} \leftarrow ADD(R_9, FP(MUL, ADD))$
10: $FP(MUL, ADD) \leftarrow MUL(R_{11}, R_6)$
11: $R_{11} \leftarrow ADD(R_{11}, FP(MUL, ADD))$
12: $R_5 \leftarrow MUL(R_{12}, R_6)$
13: $FP(MUL, ADD) \leftarrow MUL(R_{10}, R_6)$
14: $R_{17} \leftarrow ADD(R_2, FP(MUL, ADD))$
15: $R_{10} \leftarrow MUL(R_{12}, R_{20})$
16: $R_{13} \leftarrow MUL(R_{18}, R_{23})$
17: $R_{10} \leftarrow MUL(R_{10}, R_1)$
18: $FP(MUL, ADD) \leftarrow MUL(R_0, R_2)$
19: $R_9 \leftarrow ADD(R_9, FP(MUL, ADD))$
20: $FP(MUL, ADD) \leftarrow MUL(R_{12}, R_5)$
21: $R_{14} \leftarrow ADD(R_{11}, FP(MUL, ADD))$
22: $R_8 \leftarrow MUL(R_6, R_8)$
23: $R_6 \leftarrow MUL(R_{16}, R_6)$
24: $R_{12} \leftarrow MUL(R_3, R_{12})$
25: $R_{13} \leftarrow MUL(R_2, R_{13})$
26: $R_{12} \leftarrow MUL(R_{12}, R_9)$
27: $R_{11} \leftarrow MUL(R_9, R_5)$
28: $FP(MUL, ADD) \leftarrow MUL(R_9, R_{10})$
29: $R_{10} \leftarrow ADD(FP(MUL, ADD), R_9)$
30: $R_2 \leftarrow MUL(R_2, R_{13})$
31: $FP(ADD, MUL) \leftarrow ADD(R_{10}, R_{11})$
32: $R_9 \leftarrow MUL(FP(ADD, MUL), R_9)$
33: $R_{11} \leftarrow ADD(R_{11}, R_{10})$
34: $R_{11} \leftarrow ADD(R_{11}, R_{12})$
35: $R_{12} \leftarrow MUL(R_{13}, R_2)$
36: $FP(MUL, ADD) \leftarrow MUL(R_6, R_{12})$
37: $R_{12} \leftarrow ADD(R_{12}, FP(MUL, ADD))$
38: $R_{12} \leftarrow ADD(R_{12}, R_9)$
39: $FP(ADD, MUL) \leftarrow ADD(R_{13}, R_6)$
40: $R_9 \leftarrow MUL(FP(ADD, MUL), h_2)$
41: $FP(MUL, ADD) \leftarrow MUL(R_{15}, R_{13})$
42: $R_{10} \leftarrow ADD(R_{10}, FP(MUL, ADD))$
43: $FP(MUL, ADD) \leftarrow MUL(R_5, h_1)$
44: $R_9 \leftarrow ADD(R_9, FP(MUL, ADD))$
45: $R_8 \leftarrow ADD(R_9, R_8)$
46: $R_8 \leftarrow MUL(R_6, R_8)$
47: $R_{12} \leftarrow ADD(R_{12}, R_8)$
48: $R_{11} \leftarrow ADD(R_{11}, R_{12})$
49: $R_1 \leftarrow MUL(R_9, R_3)$
50: $R_{10} \leftarrow MUL(R_5, R_8)$
51: $R_3 \leftarrow MUL(R_{19}, R_{20})$
52: $R_4 \leftarrow MUL(R_{10}, h_2)$
53: $FP(MUL, ADD) \leftarrow MUL(R_{10}, h_2)$
54: $R_{10} \leftarrow ADD(R_{10}, FP(MUL, ADD))$
55: $R_8 \leftarrow ADD(R_8, R_4)$
56: $R_1 \leftarrow ADD(R_8, R_{12})$
57: $R_{10} \leftarrow ADD(R_{10}, R_2)$
58: $FP(MUL, ADD) \leftarrow MUL(R_{10}, R_{12})$
59: $R_{13} \leftarrow ADD(FP(MUL, ADD), R_{13})$
60: $FP(MUL, ADD) \leftarrow MUL(R_{10}, R_{22})$
61: $R_{10} \leftarrow ADD(FP(MUL, ADD), R_{11})$
62: $FP(MUL, ADD) \leftarrow MUL(R_{13}, h_1)$
63: $R_{10} \leftarrow ADD(R_{10}, FP(MUL, ADD))$
64: $FP(MUL, ADD) \leftarrow MUL(R_{23}, h_0)$
65: $R_{11} \leftarrow ADD(R_{13}, FP(MUL, ADD))$
```

Algorithm A.17 The modified register management of mixed addition for *recent* coordinates for an even characteristic when $h_2 \neq 0$

```
Input:  $[U_{11}, U_{10}, V_{11}, V_{10}]$,$[U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$,
        $h = h_2x^2 + h_1x + h_0$,$f = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$.
Output: $[U'_1, U'_0, V'_1, V'_0, Z', z'] = [U_{11}, U_{10}, V_{11}, V_{10}] + [U_{21}, U_{20}, V_{21}, V_{20}, Z_2, z_2]$
 1: $R_0 \leftarrow U_{21},\ R_1 \leftarrow U_{20}, R_2 \leftarrow V_{21},\ R_3 \leftarrow V_{20}, R_4 \leftarrow Z_2,\ R_5 \leftarrow z_2$

 2: $FP(MUL,\ ADD) \leftarrow MUL\,(R_4,\ U_{11})$
 3: $R_{14} \leftarrow ADD\,(FP(MUL,\ ADD),\ U_{21})$
 4: $FP(MUL,\ ADD) \leftarrow MUL\,(R_4,\ U_{10})$
 5: $R_{10} \leftarrow ADD\,(FP(MUL,\ ADD),\ U_{20})$
 6: $FP(MUL,\ ADD) \leftarrow MUL\,(R_0,\ R_{14})$
 7: $R_{15} \leftarrow ADD\,(FP(MUL,\ ADD),\ R_{10})$
 8: $R_{16} \leftarrow MUL\,(R_{10},\ R_{15})$
 9: $R_7 \leftarrow MUL\,(R_4,\ R_{14})$
10: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{17},\ R_1)$
11: $R_{16} \leftarrow ADD\,(R_{16},\ FP(MUL,\ ADD))$
12: $R_{12} \leftarrow MUL\,(R_{16},\ R_{12})$
13: $FP(MUL,\ ADD) \leftarrow MUL\,(R_3,\ R_{13})$
14: $R_1 \leftarrow ADD\,(FP(MUL,\ ADD),\ R_7)$
15: $R_{13} \leftarrow MUL\,(R_2,\ R_{13})$
16: $R_{13} \leftarrow ADD\,(R_{13},\ R_6)$
17: $FP(ADD,\ MUL) \leftarrow ADD\,(R_{15},\ R_{14})$
18: $R_{15} \leftarrow MUL\,(R_{15},\ FP(ADD,\ MUL))$
19: $R_{17} \leftarrow ADD\,(R_{17},\ R_{13})$
20: $R_{17} \leftarrow MUL\,(R_8,\ R_{17})$
21: $R_{13} \leftarrow MUL\,(R_{14},\ R_{13})$
22: $R_{13} \leftarrow MUL\,(R_1,\ R_{13})$
23: $R_{13} \leftarrow ADD\,(R_{15},\ R_{13})$
24: $R_{15} \leftarrow ADD\,(R_{17},\ R_8)$
25: $R_{16} \leftarrow MUL\,(R_{16},\ R_{15})$
26: $R_{17} \leftarrow MUL\,(R_{13},\ R_8)$
27: $R_{13} \leftarrow MUL\,(R_{13},\ R_{15})$
28: $R_7 \leftarrow MUL\,(R_{16},\ R_7)$
29: $R_6 \leftarrow MUL\,(R_{16},\ R_6)$
30: $R_{16} \leftarrow MUL\,(R_{15},\ R_{15})$
31: $R_{15} \leftarrow MUL\,(R_{15},\ R_8)$
32: $R_{10} \leftarrow MUL\,(R_{16},\ R_{10})$
33: $R_{13} \leftarrow MUL\,(R_{13},\ R_5)$
34: $R_7 \leftarrow MUL\,(R_{13},\ R_7)$
35: $R_8 \leftarrow MUL\,(R_{12},\ R_8)$
36: $R_{12} \leftarrow MUL\,(R_{12},\ R_{12})$
37: $R_{12} \leftarrow MUL\,(R_{14},\ R_{12})$
38: $R_4 \leftarrow ADD\,(R_4,\ R_5)$
39: $R_4 \leftarrow MUL\,(R_{18},\ R_4)$
40: $R_4 \leftarrow ADD\,(R_4,\ R_6)$
41: $R_5 \leftarrow MUL\,(R_8,\ h_2)$
42: $R_5 \leftarrow ADD\,(R_7,\ R_5)$
43: $R_6 \leftarrow MUL\,(R_{15},\ R_7)$
44: $R_6 \leftarrow MUL\,(R_9,\ R_6)$
45: $R_{13} \leftarrow MUL\,(R_{15},\ R_{15})$
46: $R_7 \leftarrow MUL\,(R_{13},\ R_7)$
47: $R_{14} \leftarrow MUL\,(R_{15},\ R_8)$
48: $FP(MUL,\ ADD) \leftarrow MUL\,(R_8,\ R_8)$
49: $R_{14} \leftarrow ADD\,(R_{16},\ FP(MUL,\ ADD))$
50: $R_9 \leftarrow MUL\,(R_{14},\ h_2)$
51: $FP(ADD,\ MUL) \leftarrow ADD\,(R_{11},\ R_{16})$
52: $R_9 \leftarrow MUL\,(R_0,\ FP(ADD,\ MUL))$
53: $R_5 \leftarrow ADD\,(R_5,\ R_9)$
54: $R_5 \leftarrow ADD\,(R_5,\ R_{10})$
55: $R_{10} \leftarrow MUL\,(R_{14},\ h_1)$
56: $R_5 \leftarrow ADD\,(R_5,\ R_{10})$
57: $R_5 \leftarrow ADD\,(R_5,\ R_{12})$
58: $R_4 \leftarrow ADD\,(R_4,\ R_5)$
59: $R_4 \leftarrow MUL\,(R_{13},\ R_4)$
60: $R_{10} \leftarrow MUL\,(R_{13},\ R_{14})$
61: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{14},\ h_2)$
62: $R_{12} \leftarrow ADD\,(R_{12},\ FP(MUL,\ ADD))$
63: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{14},\ h_2)$
64: $R_6 \leftarrow ADD\,(R_6,\ FP(MUL,\ ADD))$
65: $R_6 \leftarrow ADD\,(R_6,\ R_{16})$
66: $FP(MUL,\ ADD) \leftarrow MUL\,(R_6,\ R_5)$
67: $R_7 \leftarrow ADD\,(FP(MUL,\ ADD), R_7)$
68: $FP(MUL,\ ADD) \leftarrow MUL\,(R_6,\ R_{16})$
69: $R_4 \leftarrow ADD\,(R_6,\ FP(MUL,\ ADD))$
70: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{10},\ h_1)$
71: $R_4 \leftarrow ADD\,(R_4,\ FP(MUL,\ ADD))$
72: $FP(MUL,\ ADD) \leftarrow MUL\,(R_{10},\ h_0)$
73: $R_6 \leftarrow ADD\,(R_7,\ FP(MUL,\ ADD))$
```

Appendix B

VHDL Source Code

This appendix provides an example of VHDL source code of one case for performing divisor multiplication for *new weighted* coordinate for an even characteristic when $h_2 = 0$. The datapath and the control VHDL code are referring to the structure view of the *new weighted* coordinate divisor doubling and *mixed* addition datapath shown in Figure 5.2 and Figure 5.3, respectively.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.parameters.all;
package parameters is
constant M: integer := 83;
constant F: std_logic_vector(M-1 downto 0)
:=  "100"x"000000000000000000095";
end parameters;
entity NW_data_path is port(
        U1, U0, V1, V0, Z1, Z2, z_1,
        z_2, z3, z4, Din, Dout: in std_logic_vector(M-1 downto 0);
        clk, reset, start_mult, init : in std_logic;
        Sel_ADD2, Sel_MUL1, Sel_MUL2, Sel_SQR,
        Sel_Const: in std_logic;
        Sel_ADD1, Sel_cMUL: in std_logic_vector(1 downto 0);
        Sel_RA_out1, Sel_RA_out2, Sel_RA_out3, Sel_RA_out4,
        Sel_RB_out1, Sel_RB_out2, Sel_RB_out3, EN_RC
```

```
                    : in std_logic_vector(3 downto 0);
        U1o, U0o, V1o, V0o, Z1o, Z2o, z1_o, z2_o, z3o, z4o
                    : out std_logic_vector(M-1 downto 0);
        readR0, readR1, readR2, readR3, readR4, readR5, readR6,
        readR7, readR8, readR9, readR10
                    : in std_logic_vector(M-1 downto 0);
        mult_done, infinity: out std_logic );
end NW_data_path;
architecture circuit of NW_data_path is
component squarer_83 is port (
        SQR_in: in std_logic_vector(M-1 downto 0);
        SQR_out: out std_logic_vector(M-1 downto 0) );
end component;
component mult is port (
        SA_MUL, SB_MUL: in std_logic_vector (M-1 downto 0);
        clk, reset, start_mult: in std_logic;
        MUL_out: out std_logic_vector (M-1 downto 0);
        done_mult: out std_logic );
end component;
component Mux2t1 is port (
        in0 : in  STD_LOGIC_vector(M-1 downto 0);
        in1 : in  STD_LOGIC_vector(M-1 downto 0);
        Sel_mux2t1 : in Std_logic;
        Mux2t1_out : out  STD_LOGIC_vector(M-1 downto 0) );
end component;
component Mux3t1 is Port (
        in0 : in  STD_LOGIC_vector(M-1 downto 0);
        in1 : in  STD_LOGIC_vector(M-1 downto 0);
        in2 : in  STD_LOGIC_vector(M-1 downto 0);
        Sel_mux3t1 : in  STD_LOGIC_vector(1 downto 0);
        mux3t1_out : out  STD_LOGIC_vector(M-1 downto 0) );
end component;
component mux4t1 is Port (
        in0 : in  STD_LOGIC_vector(M-1 downto 0);
        in1 : in  STD_LOGIC_vector(M-1 downto 0);
        in2 : in  STD_LOGIC_vector(M-1 downto 0);
        in3 : in  STD_LOGIC_vector(M-1 downto 0);
        Sel_mux4t1 : in  STD_LOGIC_vector(1 downto 0);
        mux4t1_out : out  STD_LOGIC_vector(M-1 downto 0) );
end component;
component mux11t1 is     Port ( in0 :
        in  STD_LOGIC_vector(M-1 downto 0);
        in1 : in  STD_LOGIC_vector(M-1 downto 0);
        in2 : in  STD_LOGIC_vector(M-1 downto 0);
        in3 : in  STD_LOGIC_vector(M-1 downto 0);
        in4 : in  STD_LOGIC_vector(M-1 downto 0);
        in5 : in  STD_LOGIC_vector(M-1 downto 0);
        in6 : in  STD_LOGIC_vector(M-1 downto 0);
        in7 : in  STD_LOGIC_vector(M-1 downto 0);
        in8 : in  STD_LOGIC_vector(M-1 downto 0);
        in9 : in  STD_LOGIC_vector(M-1 downto 0);
        in10 : in  STD_LOGIC_vector(M-1 downto 0);
        Sel_mux11t1_out1 : in  STD_LOGIC_vector(3 downto 0);
        Sel_mux11t1_out2 : in STd_logic_vector(3 downto 0);
        Sel_mux11t1_out3 : in std_logic_vector(3 downto 0);
        Sel_mux11t1_out4 : in std_logic_vector(3 downto 0);
        mux11t1_out1 : out  STD_LOGIC_vector(M-1 downto 0);
        mux11t1_out2 :out std_logic_vector(M-1 downto 0);
        mux11t1_out3 :out std_logic_vector(M-1 downto 0);
        mux11t1_out4 : out std_logic_vector(M-1 downto 0) );
end component;
component mux11t3 is     Port (
        in0 : in  STD_LOGIC_vector(M-1 downto 0);
        in1 : in  STD_LOGIC_vector(M-1 downto 0);
        in2 : in  STD_LOGIC_vector(M-1 downto 0);
        in3 : in  STD_LOGIC_vector(M-1 downto 0);
        in4 : in  STD_LOGIC_vector(M-1 downto 0);
        in5 : in  STD_LOGIC_vector(M-1 downto 0);
        in6 : in  STD_LOGIC_vector(M-1 downto 0);
        in7 : in  STD_LOGIC_vector(M-1 downto 0);
        in8 : in  STD_LOGIC_vector(M-1 downto 0);
        in9 : in  STD_LOGIC_vector(M-1 downto 0);
        in10 : in  STD_LOGIC_vector(M-1 downto 0);
        Sel_mux11t3_out1 : in  STD_LOGIC_vector(3 downto 0);
        Sel_mux11t3_out2 : in STd_logic_vector(3 downto 0);
        Sel_mux11t3_out3 : in std_logic_vector(3 downto 0);
        mux11t3_out1 : out  STD_LOGIC_vector(M-1 downto 0);
        mux11t3_out2 :out std_logic_vector(M-1 downto 0);
        mux11t3_out3 :out std_logic_vector(M-1 downto 0) );
end component;
signal SA_ADD, SB_ADD, ADD_out, SA_MUL, SB_MUL, MUL_out,
cADD_in1, cADD_out, cMUL_in1,  cMUL_in2, cMUL_out,
RA_out1 ,RA_out2, RA_out3, RA_out4, RB_out1, RB_out2, RB_out3,
RC_in, SQR_in, SQR_out : std_logic_vector (M-1 downto 0);
signal R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10
                : std_logic_vector(M-1 downto 0);
signal loadR0, loadR1, loadR2, loadR3, loadR4, loadR5,
loadR6, loadR7, loadR8, loadR9, loadR10
                : std_logic; constant zero:
std_logic_vector(M-1 downto 0) := (others => '0');
constant one: std_logic_vector(M-1 downto 0)
:= conv_std_logic_vector(1, M);
subtype state_ty is std_logic_vector(19 downto 0);
constant S0 : state_ty := "00000000000000000001"
constant S1 : state_ty := "00000000000000000010"
constant S2 : state_ty := "00000000000000000100"
constant S3 : state_ty := "00000000000000001000"
constant S4 : state_ty := "00000000000000010000"
constant S5 : state_ty := "00000000000000100000"
constant S6 : state_ty := "00000000000001000000"
constant S7 : state_ty := "00000000000010000000"
constant S8 : state_ty := "00000000000100000000"
constant S9 : state_ty := "00000000001000000000"
constant S10 : state_ty := "00000000010000000000"
constant S11 : state_ty := "00000000100000000000"
```

```
 constant S12 : state_ty := "00000001000000000000"
 constant S13 : state_ty := "00000010000000000000"
 constant S14 : state_ty := "00000100000000000000"
 constant S15 : state_ty := "00001000000000000000"
 constant S16 : state_ty := "00010000000000000000"
 constant S17 : state_ty := "00100000000000000000"
 constant S18 : state_ty := "01000000000000000000"
 constant S19 : state_ty := "10000000000000000000"
 signal state : state_ty;
 begin
MUX1: Mux3t1 port map (in0 => cADD_out, in1 => ADD_out,
        in2 => cADD_out, Sel_mux3t1 => Sel_ADD1,
        mux3t1_out => SA_ADD);
MUX2: Mux2t1 port map (in0 => ADD_out,
        in1 => RA_out1, Sel_mux2t1 => Sel_MUL1,
        Mux2t1_out => SA_MUL);
MUX3: Mux4t1 port map (in0 => cMUL_out, in1 =>
        SQR_out, in2 => Din, in3 => RB_out1,  Sel_mux4t1 =>
        Sel_cMUL, mux4t1_out => cMUL_in1);
MUX4: Mux2t1 port map (in0 => RA_out2, in1 => one,
        Sel_mux2t1 => Sel_SQR, Mux2t1_out => SQR_in);
MUX5: mux11t1 port map (
        in0 => readR0, in1 => readR1, in2 => readR2,
        in3 => readR3, in4 => readR4, in5 => readR5,
        in6 => readR6, in7 => readR7, in8 => readR8,
        in9 => readR9, in10 => readR10,
        Sel_mux11t1_out1 => Sel_RA_out1, Sel_mux11t1_out2 =>
        Sel_RA_out2, Sel_mux11t1_out3 => Sel_RA_out3,
        Sel_mux11t1_out4 => Sel_RA_out4,mux11t1_out1 =>
        RA_out1, mux11t1_out2 => RA_out2, mux11t1_out3 =>
        RA_out3, mux11t1_out4 => RA_out4 );
MUX6: mux11t3 port map ( in0 => readR0, in1 => readR1,
        in2 => readR2, in3 => readR3, in4 => readR4,
        in5 => readR5, in6 => readR6, in7 => readR7,
        in8 => readR8, in9 => readR9, in10 => readR10,
        Sel_mux11t3_out1 => Sel_RB_out1,
        Sel_mux11t3_out2 => Sel_RB_out2,
        Sel_mux11t3_out3 => Sel_RB_out3,
        mux11t3_out1 => RB_out1,
        mux11t3_out2 => RB_out2, mux11t3_out3 => RB_out3);
M6: Mux2t1 port map (in0 => Din,
        in1 => RB_out2, Sel_mux2t1 =>
        Sel_ADD2, Mux2t1_out => SB_ADD);
M7: Mux2t1 port map (in0 => Din, in1 => RB_out3,
        Sel_mux2t1 => Sel_MUL2, Mux2t1_out => SB_MUL);
M8: Mux2t1 port map (in0 => SQR_out, in1 => one,
        Sel_mux2t1 => Sel_const,
    Mux2t1_out => cMUL_in2);
ADD: for i in 0 to M-1 generate
ADD_out(i) <= SA_ADD(i) XOR SB_ADD(i);
        end generate;
Bit_multiplier: mult port map (SA_MUL => SA_MUL,
        SB_MUL => SB_MUL, i_clock => i_clock,
    i_reset => i_reset, start_mult =>
        start_mult, MUL_out => MUL_out, done_mult =>
        mult_done );
cADD: for i in 0 to M-1 generate cADD_out(i) <=
        RA_out3(i) XOR one(i); end generate;
squarer: squarer_83 port map (SQR_in => SQR_in,
        SQR_out => SQR_out);
register_R0: process(i_clock) begin
if rising_edge(i_clock) then
 if i_reset = '1' then
        R0 <= U0;
 elsif loadR0 = '1' then
    R0 <= RC_in;
 end if;
end if;
end process;
register_R1: process(i_clock) begin
```

```
if rising_edge(i_clock) then
        if i_reset = '1' then
                R1 <= U1;
        elsif loadR1 = '1' then
                R1 <= RC_in;
        end if;
end if;
end process;
register_R2: process(i_clock) begin
if rising_edge(i_clock) then
        if i_reset = '1' then
                R2 <= V1;
        elsif loadR2 = '1' then
                R2 <= RC_in;
        end if;
end if;
end process;
register_R3: process(clk) begin
        if rising_edge(i_clock) then
                if i_reset = '1' then
                        R3 <= V0;
        elsif loadR3 = '1' then
                R3 <= RC_in;
        end if;
end if;
end process;
register_R4: process(clk) begin
        if rising_edge(i_clock) then
                if i_reset = '1' then
                        R4 <= Z1;
                elsif loadR4 = '1' then
                        R4 <= RC_in;
        end if;
end if;
end process;
```

```
register_R5: process(clk) begin
        if rising_edge(i_clock) then
                if i_reset = '1' then
                        R5 <= Z2;
                elsif loadR5 = '1' then
                        R5 <= RC_in;
                end if;
        end if;
end process;
register_R6: process(clk) begin
        if rising_edge(i_clock) then
                if i_reset = '1' then
                        R6 <= z_1;
                elsif loadR6 = '1' then
                        R6 <= RC_in;
                end if;
        end if;
end process;
register_R7: process(clk) begin
        if rising_edge(i_clock) then
                if i_reset = '1' then
                        R7 <= z_2;
                elsif loadR7 = '1' then
                        R7 <= RC_in;
                end if;
        end if;
end process;
register_R8: process(clk) begin
        if rising_edge(i_clock) then
                if i_reset = '1' then
                        R8 <= z3;
                elsif loadR8 = '1' then
                        R8 <= RC_in;
        end if;
end if;
end process;
register_R9: process(clk) begin
        if rising_edge(i_clock) then
                if i_reset = '1' then
                      R9 <= z4;
                elsif loadR9 = '1' then
                      R9 <= RC_in;
process(i_clock) then
        if rising_edge(i_clock) then
        if i_reset = '1' then
                state <= S0;
          else
        case state is
when S0 =>
        if in_k(M-1) = '0' then state <= S1;
        else state <= 10;
        end if;
when S1 =>
        if start_mult = '0' then state <= S2; end if;
when S2 =>
        if mult_done = '1' then state <= S3; end if;
when S2 => state <= S3
when S3 => state <= S4;
when S4 =>
        if mult_done = '1' then state <= S5; end if;
when S5 => state <= S6;
when S6 => state <= 7;
when S7 =>
        if mult_done = '1' then state <= S8; end if;
when S9 => state <= S10;
when S10 =>
        if mult_done = '1' then state <= S11;
when S11 =>
        if mult_done = '1' then state <= S12; end if;
when S12 => state <= 13;
when S13 => state <= 14;
when S14 =>
        if mult_done = '1' then state <= S15; end if;
when S15 => state <= 16;
when S16 => state <= 17;
when S17 =>
        if mult_done = '1' then state <= S18; end if;
when S18 => state <= S19;
when S19 => state <= S20;
when S20 =>
        if count < M-1 then state <= S0; end if;
        end case;
end if;
end process;
control_unit: process(i_clock, i_reset, state)
begin
case state is
when S0 to S1 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0'; Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "00";
        Sel_ADD1 <= "00";   Sel_RA <= "0101";
        Sel_RB <= "0010";    EN_RC <= "1000"; shift <= '0';
when S2 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';    Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "00";
        Sel_ADD1 <= "00";   Sel_RA <= "0101";
        Sel_RB <= "0011";    EN_RC <= "0101"; shift <= '0';
when S3 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';    Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "00";
        Sel_ADD1 <= "00";   Sel_RA <= "1000";
        Sel_RB <= "0101";  EN_RC <= "1001"; shift <= '0';
```

```
when S4 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';    Sel_SQR <= '1';
        Sel_Const <= '0'; Sel_cMUL <= "01";
        Sel_ADD1 <= "10";    Sel_RA <= "1000";
        Sel_RB <= "0110";  EN_RC <= "0100";
        shift <= '0'; done <= '1';
when S5 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';    Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "00";
        Sel_ADD1 <= "00";    Sel_RA <= "0111";
        Sel_RB <= "0100"; EN_RC <= "0011";
when S6 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';    Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "01";
        Sel_ADD1 <= "00";    Sel_RA_out1 <= "0101";
        Sel_RA_out2 <= "1010"; Sel_RA_out3 <= "0010";
        Sel_RA_out4 <= "0100";    Sel_RB <= "0100";
        EN_RC <= "0000";
when S7 =>
        Sel_ADD2 <= '1'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';    Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "11";
        Sel_ADD1 <= "10"; Sel_RA_out1 <= "1001";
        Sel_RA_out2 <= "0010"; Sel_RA_out3 <= "0100";
        Sel_RA_out4 <= "1000";    Sel_RB <= "0000";
        EN_RC <= "0010";
when S8 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0'; Sel_SQR <= '0';
        Sel_Const <= '1'; Sel_cMUL <= "00";
        Sel_ADD1 <= "00"; Sel_RA <= "0011";
        Sel_RB <= "0010"; EN_RC <= "1010";
```

```
when S9 =>
        Sel_ADD2 <= '1'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';   Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "01";
        Sel_ADD1 <= "00";   Sel_RA <= "0101";
        Sel_RB <= "0100"; EN_RC <= "0100";
when S10 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';   Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "00";
        Sel_ADD1 <= "00";   Sel_RA <= "0011";
        Sel_RB <= "0100"; EN_RC <= "0010";
when S11 =>
        Sel_ADD2 <= '1'; Sel_MUL1 <= '1';
        Sel_MUL2 <= '0';   Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "11";
        Sel_ADD1 <= "00";   Sel_RA <= "0000";
        Sel_RB <= "0000"; EN_RC <= "0000";
when S12 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '1';
        Sel_MUL2 <= '0';   Sel_SQR <= '1';
        Sel_Const <= '0'; Sel_cMUL <= "10";
        Sel_ADD1 <= "00";   Sel_RA <= "1001";
        Sel_RB <= "0010"; EN_RC <= "0100";
when S13 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';   Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "01";
        Sel_ADD1 <= "00";   Sel_RA <= "0011";
        Sel_RB <= "0010"; EN_RC <= "0011";
when S14 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';   Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "11";
        Sel_ADD1 <= "00";   Sel_RA <= "1001";
        Sel_RB <= "0111"; EN_RC <= "1001";
when S15 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';    Sel_SQR <= '1';
        Sel_Const <= '0'; Sel_cMUL <= "01";
        Sel_ADD1 <= "00";    Sel_RA <= "0100";
        Sel_RB <= "0011";  EN_RC <= "1000";
when S16 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';    Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "11";
        Sel_ADD1 <= "00";    Sel_RA <= "0110";
        Sel_RB <= "0101";  EN_RC <= "0110";
when S17 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '1';
        Sel_MUL2 <= '0';    Sel_SQR <= '1';
        Sel_Const <= '1'; Sel_cMUL <= "11";
        Sel_ADD1 <= "00";    Sel_RA <= "0100";
        Sel_RB <= "0011"; EN_RC <= "0100";
when S18 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '0';    Sel_SQR <= '0';
        Sel_Const <= '0'; Sel_cMUL <= "11";
        Sel_ADD1 <= "00";    Sel_RA <= "0100";
        Sel_RB <= "0010"; EN_RC <= "0100";
when S19 =>
        Sel_ADD2 <= '0'; Sel_MUL1 <= '1';
        Sel_MUL2 <= '0';    Sel_SQR <= '1';
        Sel_Const <= '1'; Sel_cMUL <= "11";
        Sel_ADD1 <= "00";    Sel_RA <= "1001";
        Sel_RB <= "0111"; EN_RC <= "0001";
when 20 =>
        Sel_ADD2 <= '1'; Sel_MUL1 <= '0';
        Sel_MUL2 <= '1';    Sel_SQR <= '0';
        Sel_Const <= '1'; Sel_cMUL <= "00";
```

```
         Sel_ADD1 <= "00";     Sel_RA <= "0100";
         Sel_RB <= "1000"; EN_RC <= "0010";
end case;
end circuit;
```

Appendix C

Synthesis and Simulation

RTL schematic for $\mathbb{F}_{2^{163}}$ multiplier from Mentor Graphics Precession RTL is shown in Figure C.1. This schematic was performed by creating a VHDL model of a finite field multiplier component. This model is used with the top-level component of the divisor addition datapath to form a complete divisor multiplication. Similarly, RTL schematic for $\mathbb{F}_{2^{83}}$and $\mathbb{F}_{2^{163}}$ parallel squarer are shown in Figure C.3 and Figure C.2, respectively.

Simulation waveforms from ModelSim is shown in Figure C.4. In this simulation run, the two value of the $\mathbb{F}_{2^{163}}$ multiplier were set to two random hexadecimal values or 163-bit in binary. The expected result is done at 16840 ns. The simulation was performed by creating a VHDL test bench model of a finite field multiplier. Similarly, simulation waveforms for the $\mathbb{F}_{2^{163}}$ parallel squarer from ModelSim as well as the VHDL model are shown in Figure C.5. The expected result is done at 5.932 ns which has been considered as one clock cycle. This VHDL model is used with the $\mathbb{F}_{2^{163}}$multiplier model to form a complete FFAU used in the higher-level component of the divisor addition datapath.

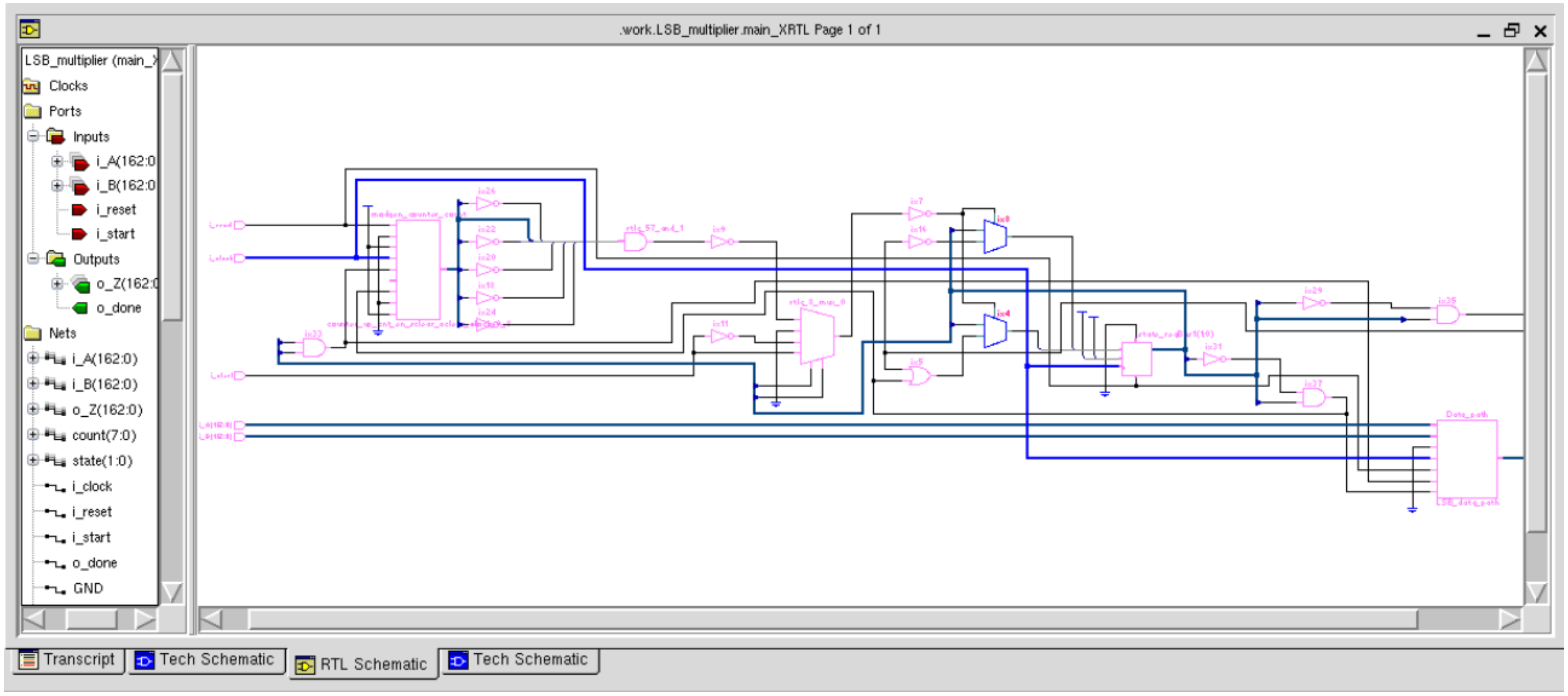


Figure C.1: **RTL schematic results after synthesis for $\mathbb{F}_{2^{163}}$ multiplier component**

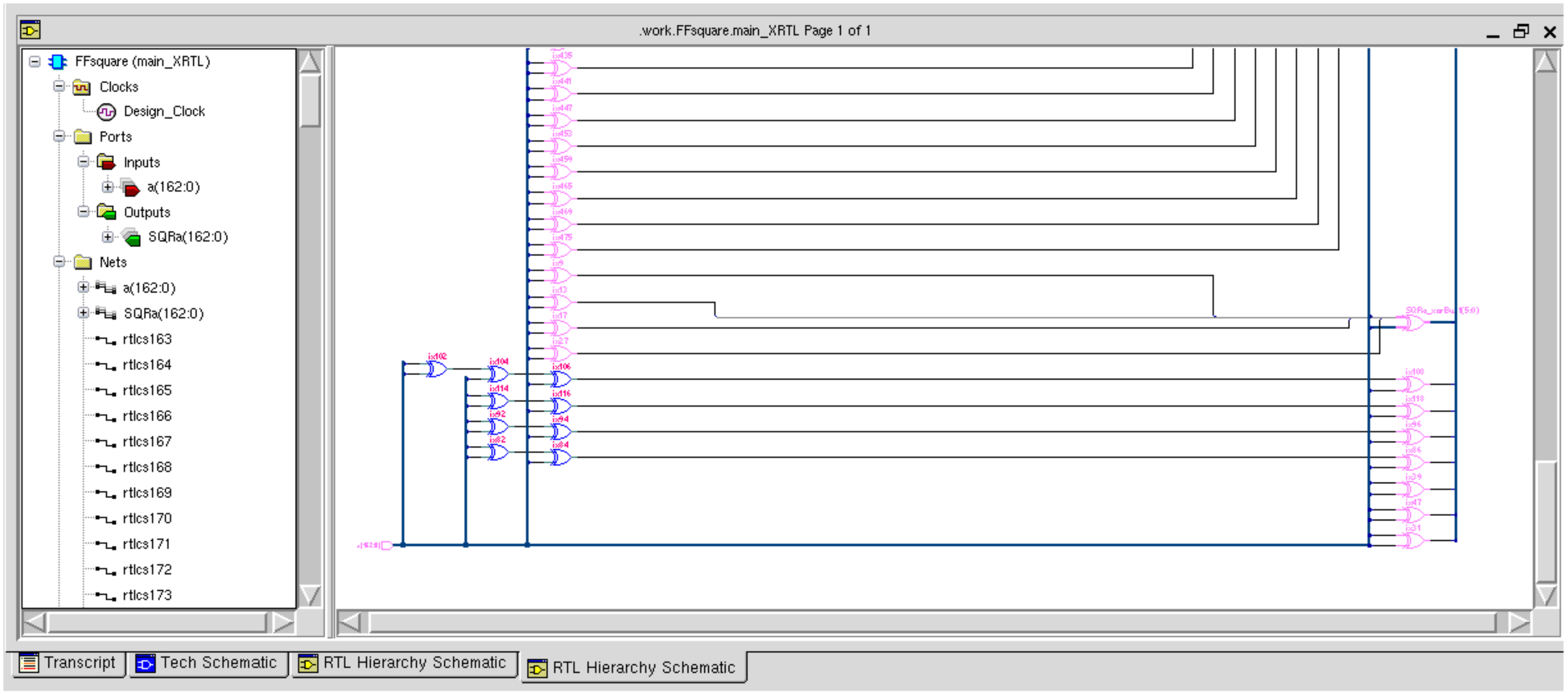
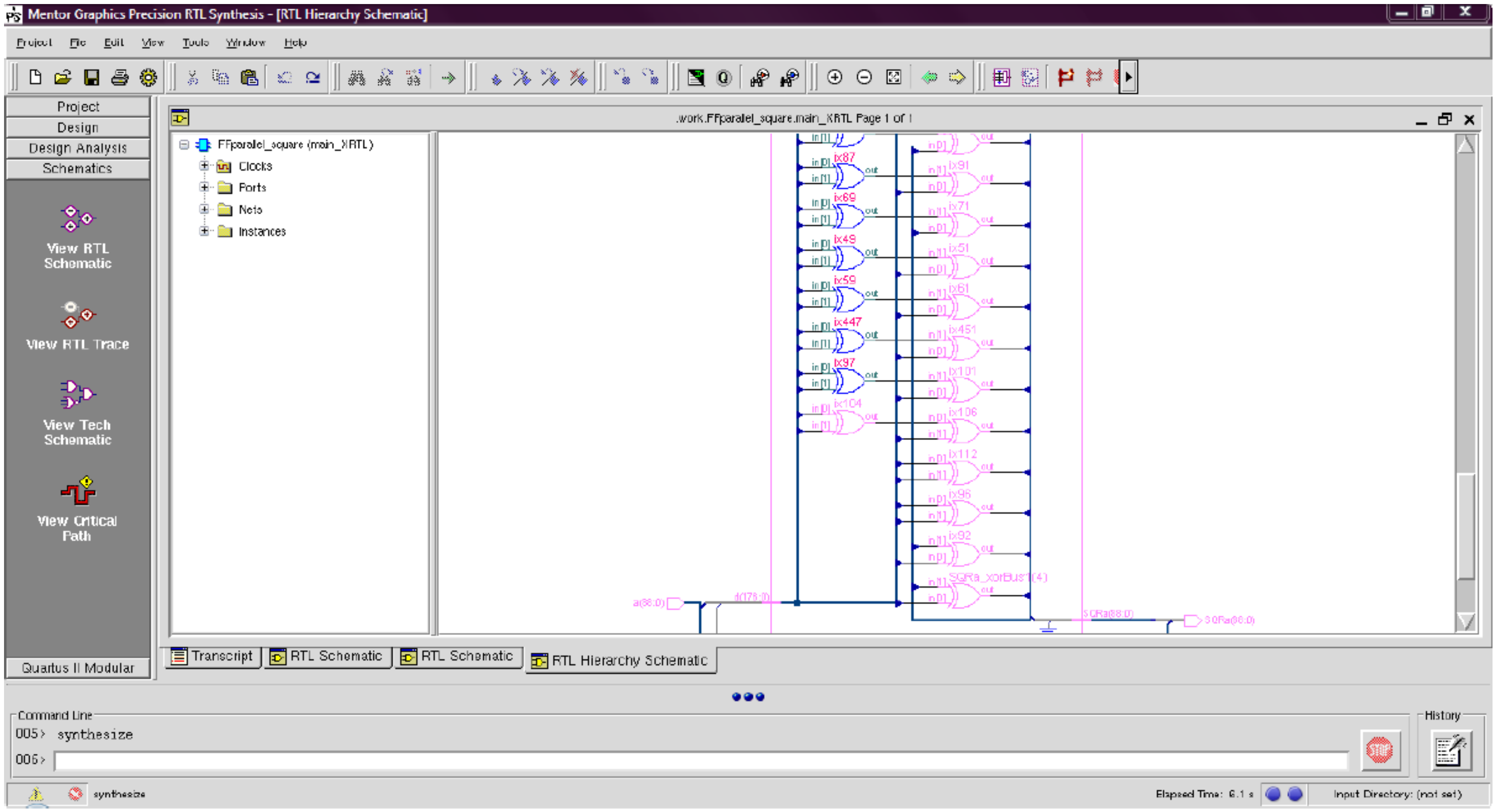
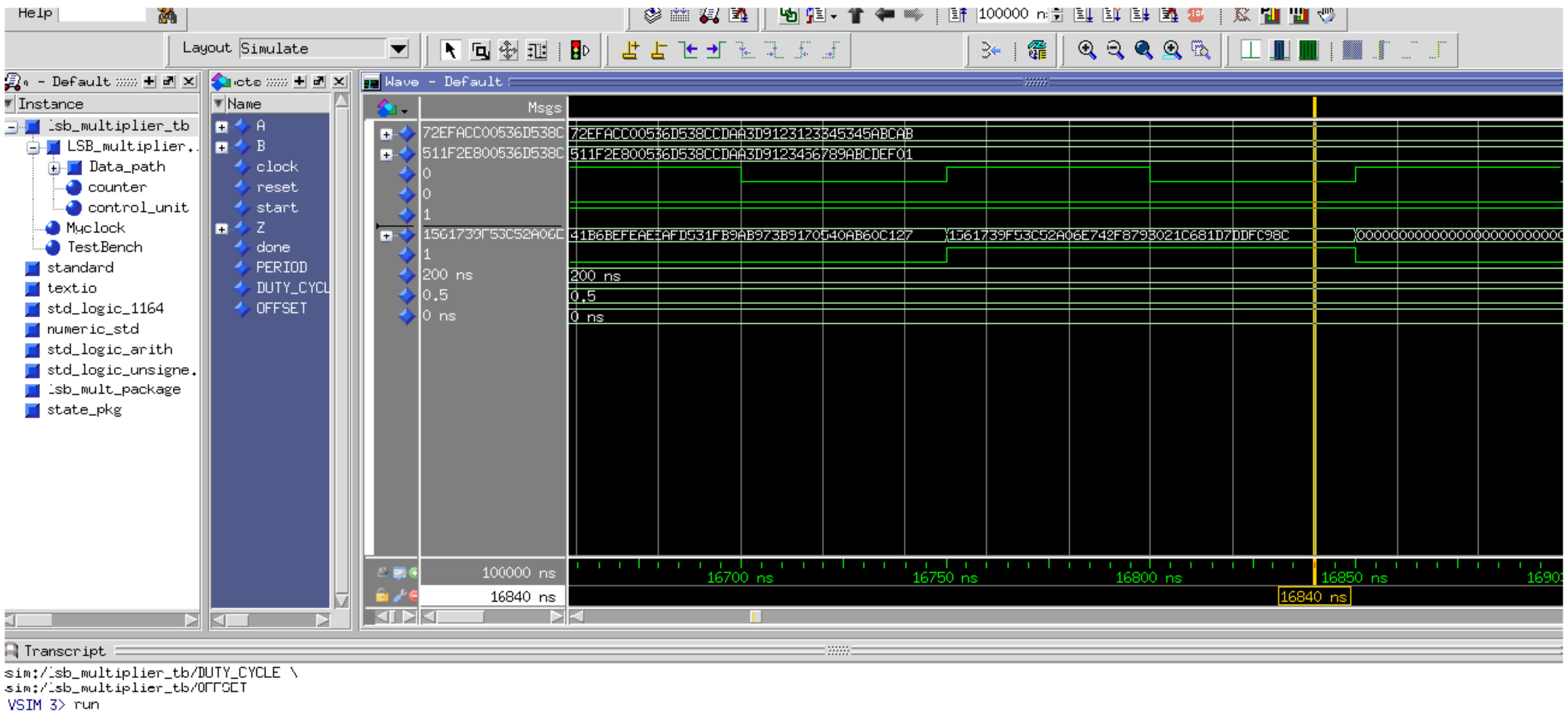


Figure C.2: **RTL schematic results after synthesis for $\mathbb{F}_{2^{163}}$parallel squarer component**



```
005> synthesize
006>
```

Figure C.3: **RTL schematic results after synthesis for $\mathbb{F}_{2^{83}}$parallel squarer component**

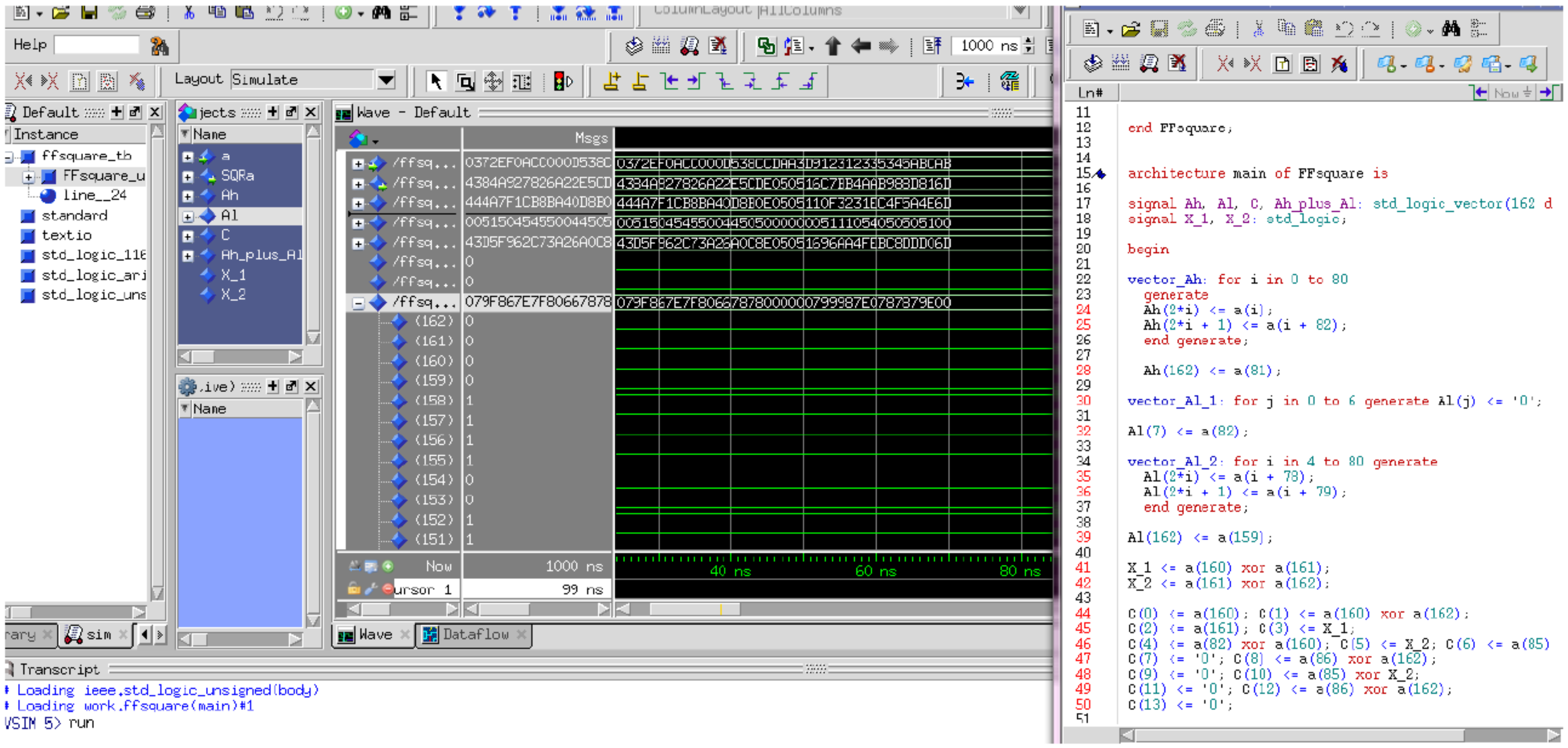


```
Instance
lsb_multiplier_tb
  LSB_multiplier..
    Data_path
    counter
    control_unit
  Myclock
  TestBench
standard
textio
std_logic_1164
numeric_std
std_logic_arith
std_logic_unsigne.
lsb_mult_package
state_pkg
```

```
Name        Msgs
A           72EFACC00536D538C   72EFACC00536D538CCDAA3D9123123345345ABCAB
B           511F2E800536D538C   511F2E800536D538CCDAA3D9123456789ABCDEF01
clock       0
reset       0
start       1
Z           1561739F53C52A06E   41B6BEFEAEIAFD531FB9AB973B9170540AB60C127   1561739F53C52A06E742F8793021C681D7DDFC98C   0000000000000000000000000
done        1
PERIOD      200 ns              200 ns
DUTY_CYCL   0.5                 0.5
OFFSET      0 ns                0 ns

100000 ns   16700 ns   16750 ns   16800 ns   16850 ns   1690
16840 ns    16840 ns
```

```
Transcript
sim:/lsb_multiplier_tb/DUTY_CYCLE \
sim:/lsb_multiplier_tb/OFFSET
VSIM 3> run
```

Figure C.4: **Simulation waveforms for $\mathbb{F}_{2^{163}}$ multiplier VHDL model**

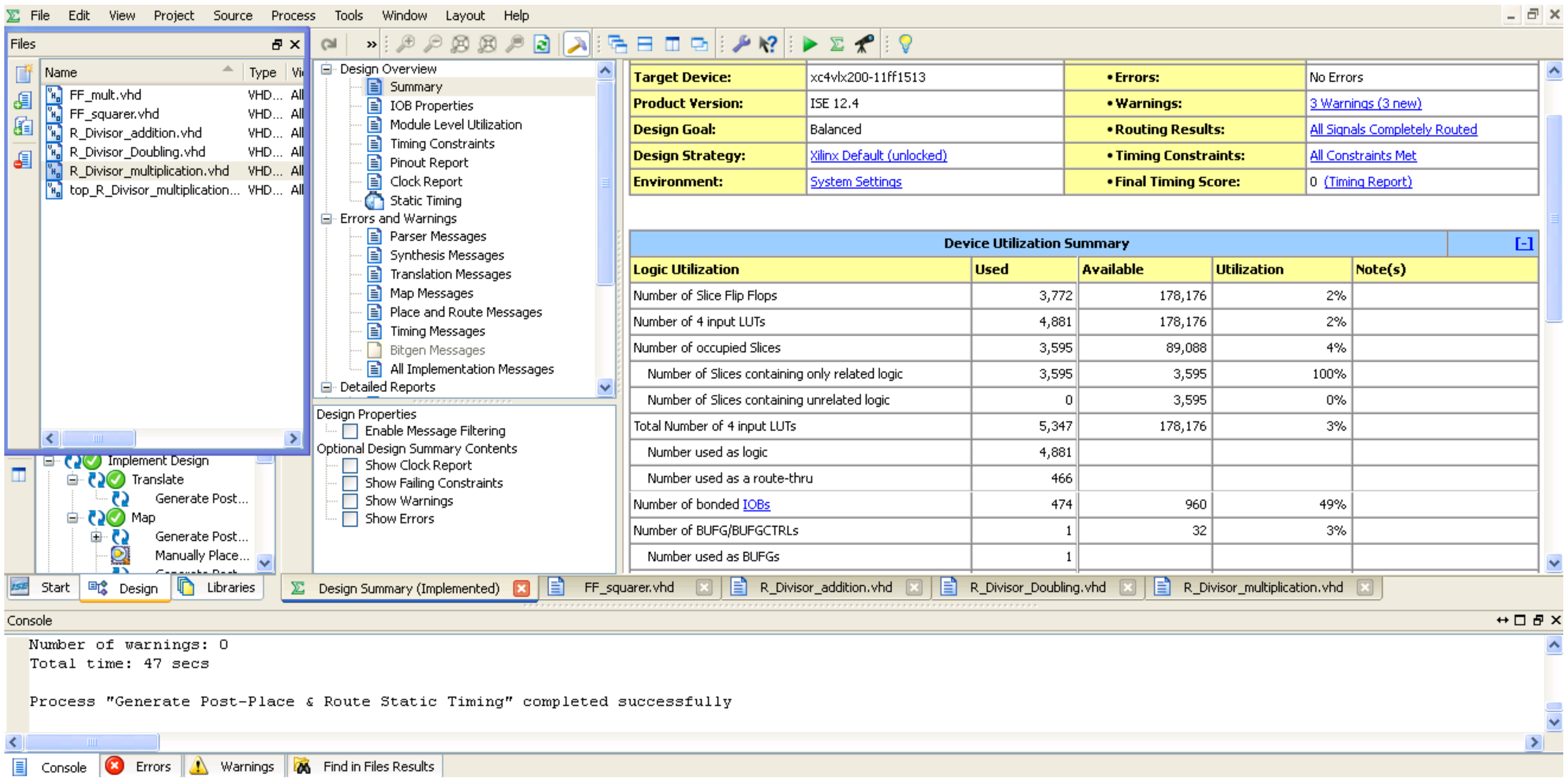


```
/ffsq...  0372EF0ACC000D538C  0372EF0ACC000D538CCDAA3D9123123353 45ABCAB
/ffsq...  4384A927826A22E5CD  4384A927826A22E5CDE050516C7BB4AAB983D816D
/ffsq...  444A7F1CB8BA40D8B0  444A7F1CB8BA40D8B0E0505110F3231EC4F5A4E6D
/ffsq...  005150454550044505  005150454550044505000000051110540505051 00
/ffsq...  43D5F962C73A26A0C8  43D5F962C73A26A0C8E05051696AA4FEBC8DDD06D
/ffsq...  0
/ffsq...  0
/ffsq...  079F867E7F80667878  079F867E7F806678780000007999 87E0787879E00
  (162)   0
  (161)   0
  (160)   0
  (159)   0
  (158)   1
  (157)   1
  (156)   1
  (155)   1
  (154)   0
  (153)   0
  (152)   1
  (151)   1
Now       1000 ns         40 ns    60 ns    80 ns
Cursor 1    99 ns
```

```
Transcript
# Loading ieee.std_logic_unsigned(body)
# Loading work.ffsquare(main)#1
VSIM 5> run
```

```vhdl
end FFsquare;

architecture main of FFsquare is

signal Ah, Al, C, Ah_plus_Al: std_logic_vector(162 d
signal X_1, X_2: std_logic;

begin

vector_Ah: for i in 0 to 80
  generate
  Ah(2*i) <= a(i);
  Ah(2*i + 1) <= a(i + 82);
  end generate;

  Ah(162) <= a(81);

vector_Al_1: for j in 0 to 6 generate Al(j) <= '0';

Al(7) <= a(82);

vector_Al_2: for i in 4 to 80 generate
  Al(2*i) <= a(i + 78);
  Al(2*i + 1) <= a(i + 79);
  end generate;

Al(162) <= a(159);

X_1 <= a(160) xor a(161);
X_2 <= a(161) xor a(162);

C(0) <= a(160); C(1) <= a(160) xor a(162);
C(2) <= a(161); C(3) <= X_1;
C(4) <= a(82) xor a(160); C(5) <= X_2; C(6) <= a(85)
C(7) <= '0'; C(8) <= a(86) xor a(162);
C(9) <= '0'; C(10) <= a(85) xor X_2;
C(11) <= '0'; C(12) <= a(86) xor a(162);
C(13) <= '0';
```

Figure C.5: **Simulation waveforms for $\mathbb{F}_{2^{163}}$ parallel squarer VHDL model**



Target Device:	xc4vlx200-11ff1513	•Errors:	No Errors
Product Version:	ISE 12.4	•Warnings:	3 Warnings (3 new)
Design Goal:	Balanced	•Routing Results:	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	•Timing Constraints:	All Constraints Met
Environment:	System Settings	•Final Timing Score:	0 (Timing Report)

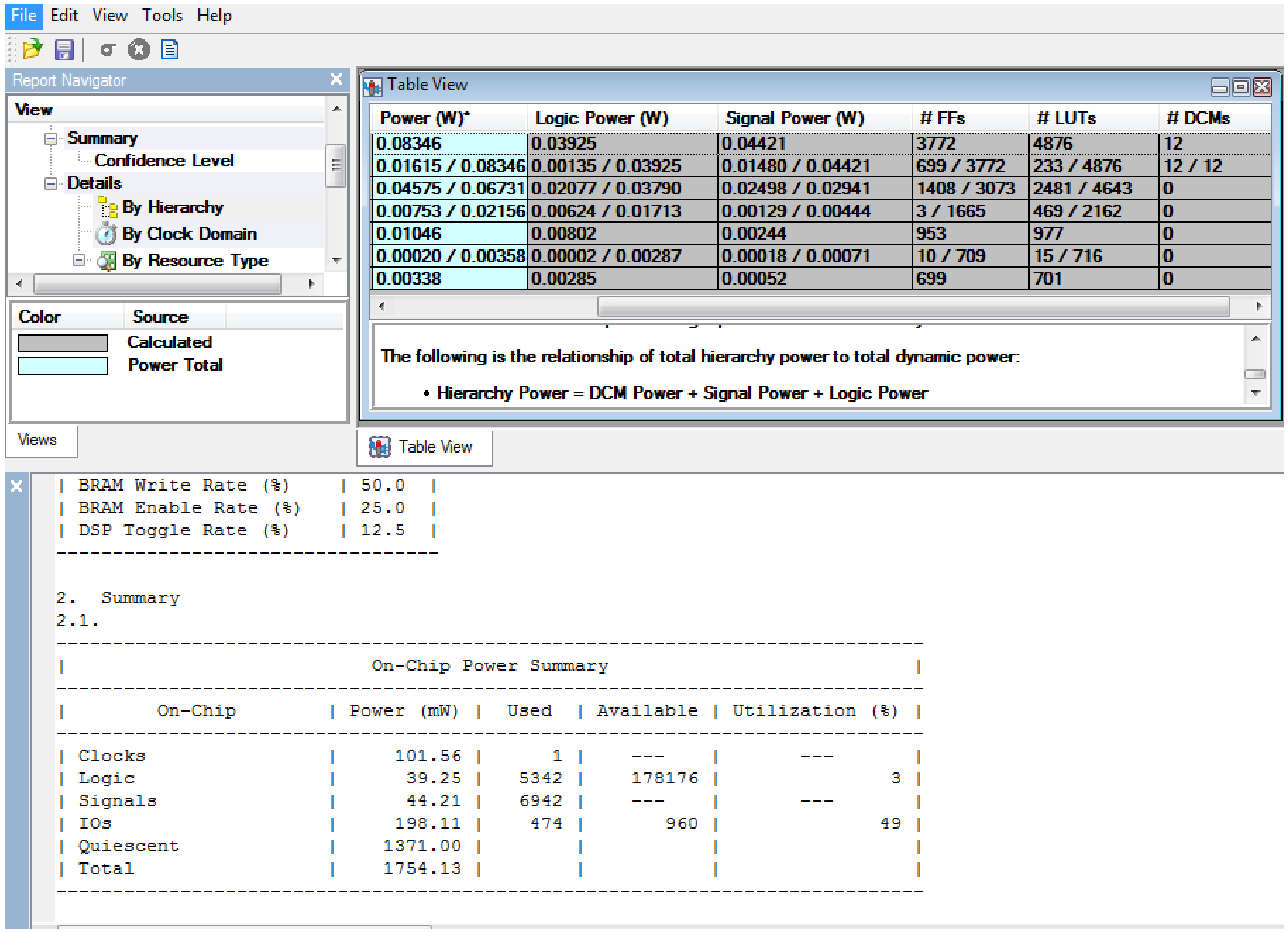
Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	3,772	178,176	2%	
Number of 4 input LUTs	4,881	178,176	2%	
Number of occupied Slices	3,595	89,088	4%	
Number of Slices containing only related logic	3,595	3,595	100%	
Number of Slices containing unrelated logic	0	3,595	0%	
Total Number of 4 input LUTs	5,347	178,176	3%	
Number used as logic	4,881			
Number used as a route-thru	466			
Number of bonded IOBs	474	960	49%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			

Name	Type
FF_mult.vhd	VHD...
FF_squarer.vhd	VHD...
R_Divisor_addition.vhd	VHD...
R_Divisor_Doubling.vhd	VHD...
R_Divisor_multiplication.vhd	VHD...
top_R_Divisor_multiplication...	VHD...

```
Number of warnings: 0
Total time: 47 secs

Process "Generate Post-Place & Route Static Timing" completed successfully
```

Figure C.6: **FPGA synthesis area results for *recent coordinates* divisor multiplication when** $h_2 \neq 0$



Color	Source
▨	Calculated
▭	Power Total

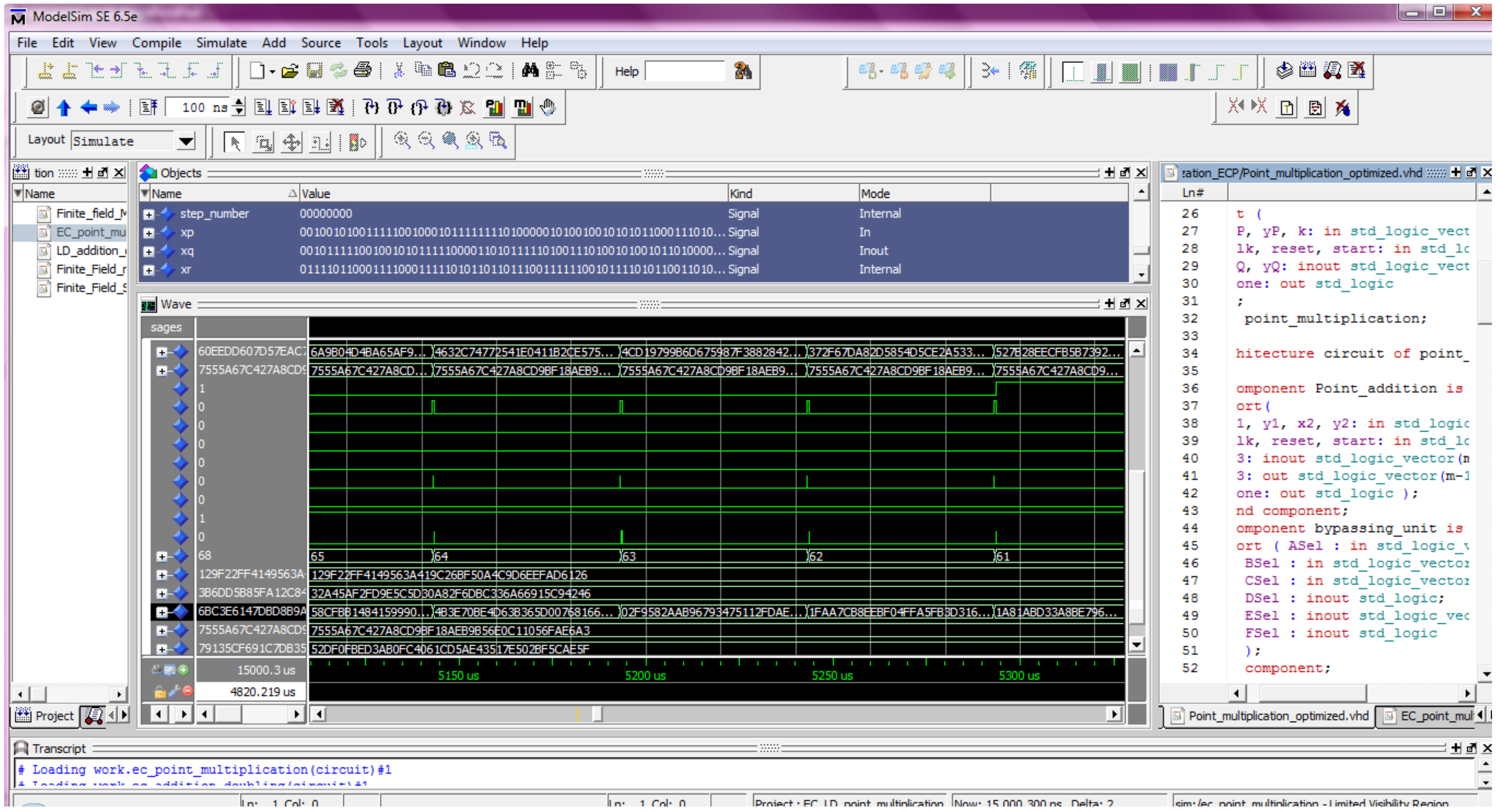
Power (W)*	Logic Power (W)	Signal Power (W)	# FFs	# LUTs	# DCMs
0.08346	0.03925	0.04421	3772	4876	12
0.01615 / 0.08346	0.00135 / 0.03925	0.01480 / 0.04421	699 / 3772	233 / 4876	12 / 12
0.04575 / 0.06731	0.02077 / 0.03790	0.02498 / 0.02941	1408 / 3073	2481 / 4643	0
0.00753 / 0.02156	0.00624 / 0.01713	0.00129 / 0.00444	3 / 1665	469 / 2162	0
0.01046	0.00802	0.00244	953	977	0
0.00020 / 0.00358	0.00002 / 0.00287	0.00018 / 0.00071	10 / 709	15 / 716	0
0.00338	0.00285	0.00052	699	701	0

The following is the relationship of total hierarchy power to total dynamic power:

- Hierarchy Power = DCM Power + Signal Power + Logic Power

```
| BRAM Write Rate (%)     | 50.0  |
| BRAM Enable Rate (%)    | 25.0  |
| DSP Toggle Rate (%)     | 12.5  |
----------------------------------

2.  Summary
2.1.
-----------------------------------------------------------------------------
|                         On-Chip Power Summary                              |
-----------------------------------------------------------------------------
|        On-Chip          | Power (mW) |  Used   | Available | Utilization (%) |
-----------------------------------------------------------------------------
| Clocks                  |     101.56 |       1 |    ---    |       ---       |
| Logic                   |      39.25 |    5342 |   178176  |               3 |
| Signals                 |      44.21 |    6942 |    ---    |       ---       |
| IOs                     |     198.11 |     474 |      960  |              49 |
| Quiescent               |    1371.00 |         |           |                 |
| Total                   |    1754.13 |         |           |                 |
-----------------------------------------------------------------------------
```

Figure C.7: **FPGA synthesis power results through X power for *recent coordinates* when $h_2 \neq 0$ at 100 MHz**



Name	Value	Kind	Mode
step_number	00000000	Signal	Internal
xp	00100101001111100100010111111111010000010100100101010110001110 10...	Signal	In
xq	00101111100100101011111000011010111110100111010010100101101 0000...	Signal	Inout
xr	01111011000111100011111010110110111001111110010111101011001101 0...	Signal	Internal

```
sages
60EEDD607D57EAC7  6A9B04D4BA65AF9...  4632C74772541E0411B2CE575...  4CD19799B6D675987F3882842...  372F67DA82D5854D5CE2A533...  527B28EECFB5B7392...
7555A67C427A8CD9  7555A67C427A8CD...  7555A67C427A8CD9BF18AEB9...  7555A67C427A8CD9BF18AEB9...  7555A67C427A8CD9BF18AEB9...  7555A67C427A8CD9...
1
0
0
0
0
0
0
1
0
68                65  64  63  62  61
129F22FF4149563A  129F22FF4149563A419C26BF50A4C9D6EEFAD6126
3B6DD5B85FA12C84  32A45AF2FD9E5C5D30A82F6DBC336A66915C94246
6BC3E6147DBD8B9A  58CFBB1484159990...  4B3E70BE4D63B365D00768166...  02F9582AAB96793475112FDAE...  1FAA7CB8EEBF04FFA5FB3D316...  1A81ABD33A8BE796...
7555A67C427A8CD9  7555A67C427A8CD9BF18AEB9B56E0C11056FAE6A3
79135CF691C7DB35  52DF0FBED3AB0FC4061CD5AE43517E502BF5CAE5F
15000.3 us        5150 us   5200 us   5250 us   5300 us
4820.219 us
```

```
t (
P, yP, k: in std_logic_vect
lk, reset, start: in std_lc
Q, yQ: inout std_logic_vect
one: out std_logic
;
 point_multiplication;

hitecture circuit of point_

omponent Point_addition is
ort(
1, y1, x2, y2: in std_logic
lk, reset, start: in std_lc
3: inout std_logic_vector(m
3: out std_logic_vector(m-1
one: out std_logic );
nd component;
omponent bypassing_unit is
ort ( ASel : in std_logic_v
 BSel : in std_logic_vector
 CSel : in std_logic_vector
 DSel : inout std_logic;
 ESel : inout std_logic_vec
 FSel : inout std_logic
 );
 component;
```

```
# Loading work.ec_point_multiplication(circuit)#1
```

```
Ln: 1 Col: 0    Ln: 1 Col: 0    Project : EC_LD_point_multiplication    Now: 15,000,300 ns  Delta: 2    sim:/ec_point_multiplication - Limited Visibility Region
```

Figure C.8: **Simulation waveforms for $\mathbb{F}_{2^{163}}$ ECC scalar multiplication VHDL model**