Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Master's Thesis

Cryptographic coprocessor

Bc. Tomáš Davidovič

Supervisor: Ing. Martin Novotný

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

January 26, 2011

Acknowledgments

I would like to thank Ing. Martin Novotný for his invaluable leadership in this project, my family for its unyielding support and the uncountable colleagues and friends who always had a kind word or two in the times of crisis. I would also like to thank the CESNET association for giving me the opportunity to evaluate this coprocessor on their Combo6X PCI board.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act §60 Zákona č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

V Praze dne 27.4. 2005 .....

Abstract

This thesis deals with a design of a versatile cryptographic coprocessor for Elliptic Curve Cryptography dedicated for cryptographic operations over binary finite field, $GF(2^m)$. The processor can work with (almost) any binary finite field of order (cardinality, number of elements) between 2^2 and 2^1000, can operate over either affine or projective coordinates and can use either polynomial basis or normal basis represenatation of field elements. The change of coordinate system is realized by a replacement of a controllers microprogram. The change of basis is done by replacement of appropriate arithmetic units and a minor change in a microprogram.

We use a Combo6X card as an implementation platform. We compare various processor configurations in area, frequency and clock cycles spent on the basic operation, scalar multiplication of a point on a curve. We also evaluate total time per single multiplication to determine whether PCI bus latencies prevent us from using Combo6X as a dedicated accelerator.

Abstrakt

Tato práce se zabývá vývojem univerzálního kryptografického koprocesoru pro kryptografii eliptických křivek, určeného pro kryptografické operace nad binarním konečným tělesem $GF(2^m)$. Procesor umí pracovat se (skoro) libovolným konečným tělesem (mohutnost, počet prvků) mezi 2^2 a 2^{1000}, umí operovat s jak afinními tak projektivními souřadnicemi a umí používat jak polynomiální tak normální bázi k reprezentaci prvků konečného tělesa. Změna souřadného systému je realizována změnou mikroprogramu řadiče. Změna báze je prováděna výměnou příslušných aritmetických jednotek a drobnými změnami mikroprogramu.

Jako implementační platformu používáme kartu Combo6X. Porovnáváme jednotlivé konfigurace procesoru v ploše, frekvenci a počtu hodinových cyklů, které zabere základní operace, skalární násobek bodu na křivce. Dále vyhodnocujeme celkévý čas spotřebovaný na jedno násobení, abychom určili zda-li nám latence PCI sběrnice brání v použití Combo6X jako dedikovaného akcelerátoru.

Contents

List of Figures

List of Tables

1 Introduction

The requirement to keep messages secret is almost as old as messages themselves. And where messengers could not be trusted to deliver the messages safely, means had to be devised to prevent unauthorized persons from reading the message contents. Thus ciphers and cryptography have been born.

Modern cryptography uses mathematics and computers for message encryption and decryption. It divides ciphers into two main categories:

- Symmetric ciphers
- Asymmetric ciphers

Symmetric ciphers, typical example is AES[1], use the same key for encryption and decryption. These ciphers have fast encryption and decryption, are very strong and therefore perfectly suitable for transferring messages between two people. However, they still have the main problem of all ciphers used throughout the centuries. If we have an unbreakable cipher, how do we tell our desired recipient how to decipher it, without the risk of telling the same thing to others?

This problem is answered by the second category of ciphers, asymmetric ciphers. Those have two different keys, one for encryption and the other for decryption. Two typical usages are:

- Release the encryption key as *public key* and keep the decryption key as *private key*. This way anyone can send us a message, for example key for a symmetric cipher, and nobody else can read the messages.
- Release the decryption key and keep the encryption key. This way we can encrypt hash of our message and everyone can check, by decrypting it, that it indeed does match the message. However, nobody can forge our message and generate a new encrypted hash that would be correctly decrypted by the decryption key. This method is known as digital signature (DSA).

For this versatility of asymmetric ciphers we pay with considerably longer computation times when using the ciphers, so they are not really suitable for general data transfers. We therefore mainly use them for the aforementioned key exchange and digital signature tasks.

Probably the best known asymmetric cipher is RSA, which is based on integer factorization problem. This problem states, that given a product of two prime numbers, it is very hard to deduce the two original prime numbers. This problem might seem to be fairly easy at first, but using large numbers has proven to be very hard. The longest key deciphered in the RSA Factoring Challenge was RSA-200 (200 decimal, 663 binary digits) and it took the solvers equivalent of 55 years of a single 2.2 GHz Opteron computer time [9]. Current standard RSA key length is typically 1024 bits and the required time to decipher roughly doubles with every binary digit we add.

[1] Advanced Encryption Standard

Elliptic Curves can also be used to build asymmetric ciphers. This relatively new field of cryptography, called Elliptic Curve Cryptography (ECC), is based on Elliptic Curve Discrete Logarithm Problem (ECDLP). The basic operation here is not integer multiplication and factorization, but scalar multiplication of a point on an elliptic curve. This cipher has, for the same given cryptographic strength, much shorter key sizes. According to [14], ECC key size 160b is equivalent to RSA key size 1024b.

Asymmetric ciphers are used not only on desktop computers, but also in RFID[2] chips, credit cards, mobile phones etc. There the 1024b RSA key length requires considerable amount of memory and registers, which are not always readily available. On the other hand, ECC requires less than sixth of that for the same strength of the cipher. We are therefore presented with an opportunity to use ECC to build smaller cryptographic hardware while retaining the high security of 1024b RSA.

The goal of this thesis was to design and implement a special cryptographic coprocessor for Elliptic Curve Cryptography that would allow easy comparisons of several ECC variants. We also aimed to implement this coprocessor on Combo6X PCI board and evaluate its function as an accelerator for desktop computers.

In Chapter 2 we will provide a brief introduction into the mathematics underlying ECC and give and, based on that, give a more detailed specification of our goals. In Chapter 3 we will describe the Combo6X card, provide basic design decisions and compare with previous work [2]. In Chapter 4 we provide detailed description of our implementation and various decisions in the process. Chapter 5 describes required support tools, in Chapter 6 we describe our tests and provide results and measurements. In Chapter 7 we conclude and suggest future work on the topic.

[2]Radio-frequency identification

2 Elliptic Curve Cryptography basics

In this chapter we will first describe several definitions and principles about elliptic curves in general. We will then focus on their representation that is used in asymmetric cryptography, explain the basic operations and determine what requirements it puts on our coprocessor. Content of this chapter is based mostly on [2], [4], [15] and [13].

2.1 Basic elliptic curve mathematics

Elliptic curve E over the real number set is a set of points that fulfill the *Weierstrass equation*:

$$y^2 = x^3 + ax + b \tag{2.1}$$

where x and y are real numbers representing a point on the curve, a and b are real numbers, defining parameters of the curve.

If the $x^3 + ax + b$ term cannot be decomposed, i.e. condition $4a^3 + 27b^2 \neq 0$ is satisfied, the elliptic curve E will form a group.

The group consists of points satisfying the Weierstrass equation and of an additional element called the *point at infinity* (denoted O). The points other than O are called *finite* points. The number of points on E (including O) is called the *order of E*.

The basic operation on the points of an elliptic curve is *addition*. This operation can be described geometrically as follows. First we define *inverse* of a point $P = (x, y)$ to be $P = (x, -y)$, i.e. the point is mirrored by the x axis. Then the sum of $P + Q$ is the point R with the property that P, Q and $-R$ lie on a common line (Figure 2.1).

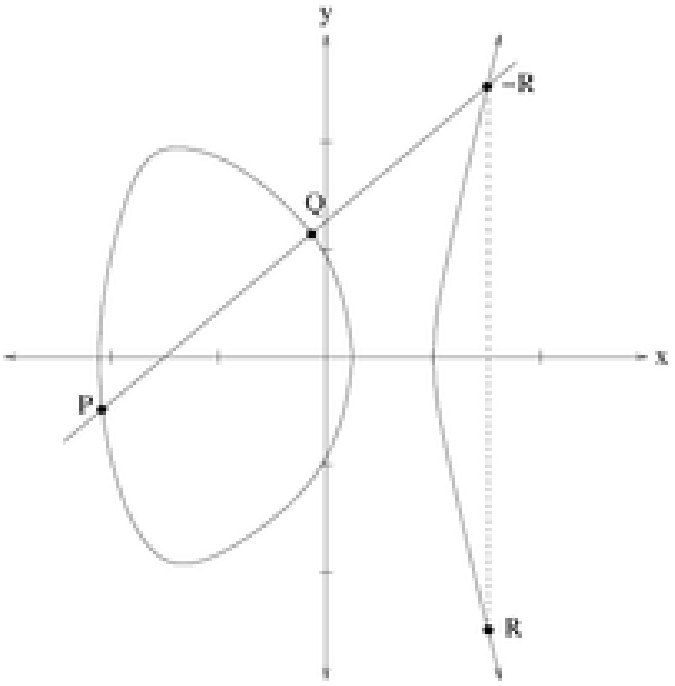

Figure 2.1: Sum of two different points on an elliptic curve, source [2]

The point at infinity O plays a role analogous to the number *0* in ordinary addition. Thus:

$$\begin{aligned} P + O &= P \\ P + (-P) &= O \end{aligned} \tag{2.2}$$

for all points P.

The last possibility not shown is adding a point to itself. This is similar to sum of two different points, only the common line contains only points P, $-R$ and no other point of the curve. Therefore, the line has to be tangent to the curve.

2.2 Mathematics for Elliptic Curve Cryptography

So far we have described the elliptic curves in their general, geometric representation. However, using this representation on computers requires floating numbers and introduces rounding error. Cryptography requires the computations to be both fast and precise. The standard [4] therefore defines elliptic curves for cryptography over finite fields.

First of the finite fields is $GF(p)$, where p is a prime number. Here the elements are integers from range $[0, p)$ and all arithmetic is done modulo p. Elliptic curve over $GF(p)$ is defined as:

$$y^2 \equiv x^3 + ax + b \; (mod\, p) \tag{2.3}$$

The second option is $GF(2^m)$, where m is the key length. Elements of this field are binary numbers with m-bit length, which is perfectly suitable to fully use our m-bit registers. We use this second representation and we use m to denote a variable defining the key length throughout this document. Elliptic curves over $GF(2^m)$ are defined by equation:

$$y^2 + xy = x^3 + ax + b \tag{2.4}$$

Unlike the prime field case, there are many common representations for binary finite fields. We focus our attention only on the following two representations:

- Polynomial basis – The finite field is generated by an appropriate irreducible field polynomial of degree m. Each element is then represented as $\alpha_{m-1}x^{m-1} + \alpha_{m-2}x^{m-2} + \ldots + \alpha_1 x + \alpha_0 1$
- Normal basis – Gaussian normal basis used in our coprocessor requires $m \; mod \; 8 \neq 0$. When this is satisfied, normal basis can be used and each element is represented as $\alpha_{m-1}x^{2^{m-1}} + \alpha_{m-2}x^{2^{m-2}} + \ldots + \alpha_1 x^2 + \alpha_0 x$

The last thing we need to define to have the complete set of operations required for Elliptic Curve Cryptography is scalar multiplication of a point. The definition is:

$$Q = k \cdot P = \underbrace{P + P + \cdots + P}_{k} \tag{2.5}$$

where k is a natural number. While its length is not specified, for convenience it is usually of the same bit length as the finite field elements.

2.3 Elliptic curves in cryptography

Now we can describe how elliptic curves are used in cryptography. As was stated before, the ECC is based on Elliptic Curve Discrete Logarithm Problem. It can be described as follows (quoted from [18]):

"Let E be an elliptic curve defined over a finite field F and give a pair of point [P, $m \cdot P$] where P is a point of order n. The ECDLP is to find the integer m where $0 \leq m \leq n-1$."

We will now give an example of how this can be used by Alice and Bob to agree upon a common key for a symmetric cipher. This example is, with slight modifications, taken from [2].

*"Let us assume we have a publicly defined elliptic curve **E** and a point **P** on this curve. If Alice wants to send a message to Bob, she will do the following. She will send Bob her public key. This key is a point **A**, from equation $A = a \cdot P$, where the integer **a** is her private key.*

*Bob will in return send her his public key, point **B**, computed in a similar fashion from $B = b \cdot P$, where the integer **b** is Bob's private key.*

*Alice, upon receiving the point **B**, can compute point $S = a \cdot B$ and use it as a key for a symmetric cipher. Bob will compute the same point **S** as $S = b \cdot A$ because, after a substitution, both equations give us $S = a \cdot b \cdot P$."*

We therefore see that if our coprocessor computes a scalar multiplication of a point on an elliptic curve, it is sufficient to directly offer some basic cryptographic functionality without any additional algorithms.

2.4 Impact on hardware design

We have concluded that it will be sufficient if our coprocessor implements only the basic operation of ECC, the scalar multiplication of a point ($Q = k \cdot P$) on an elliptic curve. We will apply the add-and-double algorithm to compute this multiplication, because adding P k-times would be extremely costly and de facto equivalent to computing the k from P and Q.

```
Q = O
While i in 0 to m-1
   Q = 2*Q
   if ki = 1 then
      Q = Q+P
```

Figure 2.2: Add-and-double algorithm for scalar multiplication

The add-and-double algorithm (Horner scheme) is depicted on Figure 2.2. The scalar k is used in its binary representation and k_i denotes the i^{th} binary digit with i between *0* and $m-1$. We see that to implement the scalar multiplication we need only general point addition or, more specifically, point doubling and general point addition.

Before we introduce algorithms used for point addition we have to introduce an alternative representation of points on elliptic curve. There are two coordinate systems:

- Affine coordinates – Coordinates we have considered so far, each point represented

1. If $P_0 = \circ$, then output $P_2 \leftarrow P_1$ and stop
2. If $P_1 = \circ$, then output $P_2 \leftarrow P_0$ and stop
3. If $x_0 \neq x_1$ then
   - 3.1 set $\lambda \leftarrow (y_0 + y_1) / (x_0 + x_1)$
   - 3.2 set $x_2 \leftarrow a + \lambda^2 + \lambda + x_0 + x_1$
   - 3.3 go to step 7
4. If $y_0 \neq y_1$ then output $P_2 \leftarrow \circ$ and stop
5. If $x_1 = 0$ then output $P_2 \leftarrow \circ$ and stop
6. Set
   - 6.1 $\lambda \leftarrow x_1 + y_1 / x_1$
   - 6.2 $x_2 \leftarrow a + \lambda^2 + \lambda$
7. $y_2 \leftarrow (x_1 + x_2)\,\lambda + x_2 + y_1$
8. $P_2 \leftarrow (x_2, y_2)$

Figure 2.3: Algorithm for point addition in affine coordinates, taken from [4]

by two field elements (x, y).

- Projective coordinates – This coordinate system uses three elements to represent a point (x, y, z). The conversion from affine to projective coordinates is $(x, y) \Rightarrow (x, y, 1)$. The conversion from projective to affine is $(x, y, z) \Rightarrow \left(\frac{x}{z^2}, \frac{y}{z^3}\right)$. From this it is obvious, that each point has several representations in the projective coordinate system. Point at infinity is represented by coordinates (1, 1, 0).

2.4.1 Affine coordinates

Algorithm in Figure 2.3 describes addition algorithm for points in affine coordinates. Ignoring the special cases when we are adding a point at infinity or the result is point at infinity, we can see two distinct paths through the algorithm.

When the two points are different we go through steps 3-3.3, 7 and 8. When the two points are equal, i.e. when doubling, we go through steps 6-8. In both paths we can see that the operations on field elements are: comparison, addition, squaring, multiplication and division.

2.4.2 Projective coordinates

Lets now consider projective coordinate algorithms for point addition and point doubling (Figure 2.4, Figure 2.5). We immediately notice there are two distinct algorithms, one for point doubling and the other for arbitrary point addition. It should be noted that when the point addition algorithm returns (0, 0, 0) in step 14, it is a signal that point doubling should be used instead. We also notice that, unlike with affine coordinates, there is no division in the algorithms.

On the other hand, we need many more temporary variables (up to nine, compared to one needed in affine coordinates algorithm) and a lot of multiplications. Using these algorithms, we first convert P to projective coordinates, do the scalar multiplication and

1. $T_1 \leftarrow X_1$
2. $T_2 \leftarrow Y_1$
3. $T_3 \leftarrow Z_1$
4. $T_4 \leftarrow c$
5. If $T_1 = 0$ or $T_3 = 0$ then output (1, 1, 0) and stop.
6. $T_2 \leftarrow T_2 \times T_3$
7. $T_3 \leftarrow T_3^2$
8. $T_4 \leftarrow T_3 \times T_4$
9. $T_3 \leftarrow T_1 \times T_3$
10. $T_2 \leftarrow T_2 + T_3$
11. $T_4 \leftarrow T_1 + T_4$
12. $T_4 \leftarrow T_4^2$
13. $T_4 \leftarrow T_4^2$
14. $T_1 \leftarrow T_1^2$
15. $T_2 \leftarrow T_1 + T_2$
16. $T_2 \leftarrow T_2 \times T_4$
17. $T_1 \leftarrow T_1^2$
18. $T_1 \leftarrow T_1 \times T_3$
19. $T_2 \leftarrow T_1 + T_2$
20. $T_1 \leftarrow T_4$
21. $X_2 \leftarrow T_1$
22. $Y_2 \leftarrow T_2$
23. $Z_2 \leftarrow T_3$

Figure 2.4: Algorithm for point doubling in projective coordinates, taken from [4]

at the end convert back to affine coordinates. We can see that in the point addition, we always add the unmodified point P, i.e. point with $(x, y, 1)$ coordinates. Therefore we can safely ignore steps 7 and 21, because the point P will never have $z \neq 1$. The last thing we notice is a parameter c in step 4 of the doubling algorithm. We compute $c = b^{2^{m-2}}$, therefore $b = c^4$.

To conclude, we need:

- Several m-bit registers for temporary variables
- Field element addition
- Field element squaring – can be implemented by multiplication or a special circuit
- Field element division (or inversion)
- Field element multiplication

Our ultimate goal is to have a coprocessor that can be used to evaluate various approaches to elliptic curve cryptography. To implement different algorithms (affine and projective coordinates) we will use micro-programmable controller. We also want all the units to be either universal, or easily interchangeable.

Addition is very simple and for both bases it is a bitwise XOR. Squaring could be performed in a multiplication unit, but in both bases there is a simple, purely combinational, circuit that performs the same operation. These circuits are however different for polynomial and normal basis, so this unit has to be interchangeable. Multiplication is also specific to each base. Division is also very different. In polynomial basis, we can use Extended Euclidean Algorithm (EEA) to directly divide two field polynomials. In normal basis, we have to use an Itoh-Teechai-Tsujii inversion algorithm described in [17] and [4]. This is however significantly more expensive than the EEA division, so projective coordinates with their two divisions (used only to transform from projective to affine coordinates) per scalar multiplication are of a great interest when considering normal basis.

1. $T_1 \leftarrow X_0$
2. $T_2 \leftarrow Y_0$
3. $T_3 \leftarrow Z_0$
4. $T_4 \leftarrow X_1$
5. $T_5 \leftarrow Y_1$
6. If $a \neq 0$ then
   $T_9 \leftarrow a$
7. If $Z_1 \neq 1$ then
   $T_6 \leftarrow Z_1$
   $T_7 \leftarrow T_6^2$
   $T_1 \leftarrow T_1 \times T_7$
   $T_7 \leftarrow T_6 \times T_7$
   $T_2 \leftarrow T_2 \times T_7$
8. $T_7 \leftarrow T_3^2$
9. $T_8 \leftarrow T_4 \times T_7$
10. $T_1 \leftarrow T_1 + T_8$
11. $T_7 \leftarrow T_3 \times T_7$
12. $T_8 \leftarrow T_5 \times T_7$
13. $T_2 \leftarrow T_2 + T_8$
14. If $T_1 = 0$ then
    If $T_2 = 0$ then output (0, 0, 0) and stop
    else output (1, 1, 0) and stop
15. $T_4 \leftarrow T_2 \times T_4$
16. $T_3 \leftarrow T_1 \times T_3$
17. $T_5 \leftarrow T_3 \times T_5$
18. $T_4 \leftarrow T_4 + T_5$
19. $T_5 \leftarrow T_3^2$
20. $T_7 \leftarrow T_4 \times T_5$
21. If $Z_1 \neq 1$ then
    $T_3 \leftarrow T_3 \times T_6$
22. $T_4 \leftarrow T_2 + T_3$
23. $T_2 \leftarrow T_2 \times T_4$
24. $T_5 \leftarrow T_1^2$
25. $T_1 \leftarrow T_1 \times T_5$
26. If $a \neq 0$ then
    $T_8 \leftarrow T_3^2$
    $T_9 \leftarrow T_8 \times T_9$
    $T_1 \leftarrow T_1 + T_9$
27. $T_1 \leftarrow T_1 + T_2$
28. $T_4 \leftarrow T_1 \times T_4$
29. $T_2 \leftarrow T_4 + T_7$
30. $X_2 \leftarrow T_1$
31. $Y_2 \leftarrow T_2$
32. $Z_2 \leftarrow T_3$

Figure 2.5: Algorithm for point addition in projective coordinates, taken from [4]

2.5 Goals

We have introduced two representations of an element in $GF(2^m)$, the normal and polynomial bases. We also introduced two different representations of a point on an elliptic curve, affine and projective coordinates.

The main goal of our project is to allow coprocessor reconfiguration in as contained manner as possible. The ideal state we strive to achieve is to have:

- A single boolean generic that will switch between polynomial and normal basis.
- A single integer variable that will allow setting key length in a reasonable range of 2-1000 bits

We want to limit usage of external tools as much as possible, having all required information generated by the synthesis process or, where not possible, pre-generated for the given range (e.g., irreducible field polynomials). The one obvious exception in this effort is compilation of code for the microcontroller. We choose Java as the ideal platform-independent programming language for this task.

3 Implementation platform

In this chapter we first describe our chosen implementation platform Combo6X and options it can provide us.

Combo6X is a PCI card with a VirtexII Pro FPGA[1] chip on it. It has been designed for the Liberouter project [16]. In its typical usage in networking applications, the Combo6X card is accompanied by a so-called interface card (Figure 3.1, Figure 3.2).

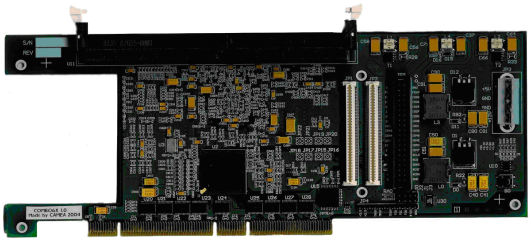

Figure 3.1: Combo6X, source [16]

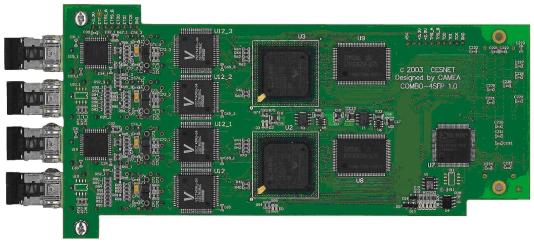

Figure 3.2: Interface card Combo-4SFP, source [16]

The Combo6X card provides communication with the rest of PC over PCI-X (66MHz/64b standard, backwards compatible with PCI 33MHz/32b we use) through a PCI bridge implemented in a dedicated FPGA with a special firmware. It also provides high computing power (VirtexII Pro FPGA), DDR and CAM memories and connection to interface cards.

There are several types of interface cards. They typically contain another Virtex-family FPGA, memories and physical network interface for either optical or metallic network cables. We do not use any such card for our project, but should we decide to use our design for automatic key exchange, then the interface cards would be used.

Our project requires only the base Combo6X card and on it only the two FPGAs, the PCI bridge and VirtexII Pro. The PCI bridge contains a "factory" firmware, is dedicated only to communication with the bus and we shall not discuss it further. We will implement our design in the main FPGA. This FPGA can be easily configured via supplied Linux tools. Neither this FPGA nor the board itself contains non-volatile memory to store the FPGA configuration, which means it is necessary to load our design after each restart of the host PC. However this apparent disadvantage proves to be a huge advantage once we load a design that violates timing on the PCI bus and freezes the whole system. It is then sufficient to simply reboot and all it back to normal, with no need to dismantle the PC and reprogram the Combo6X board externally.

As was said before, we use only PCI interface and will now briefly describe the mechanisms for connecting designs (modules) in VirtexII Pro to PCI bus.

3.1 Local bus

Communication between PCI bridge and VirtexII Pro FPGA is provided by PLX protocol. However, this protocol is not very user-friendly and therefore the basic package of the Liberouter project offers a LocalBus component that provides a much more friendly

[1] Field-programmable Gate Array

interface. This bus allows us to connect each design module to PCI via a protocol very similar to a protocol used for communication with BlockRAMs. The basic structure of this communication hierarchy is apparent from Figure 3.3.

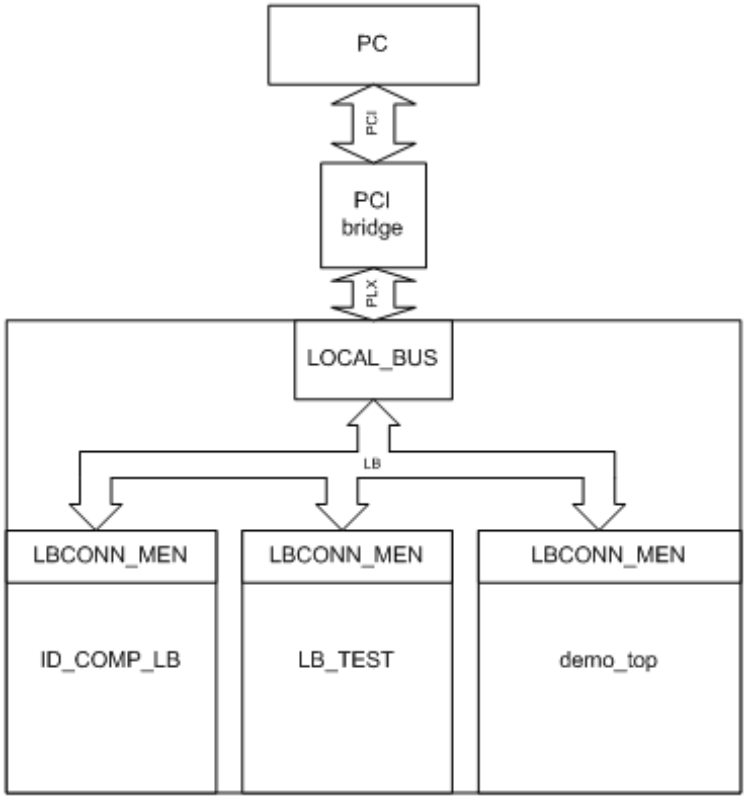

Figure 3.3: Structure of communication hierarchy in Combo6X test design

The LOCAL_BUS component negotiates communication between the PCI bridge and other LocalBus components. There are also LocalBus components that provide communication with interface cards, but we do not use these in our design and thus they are not depicted on Figure 3.3.

We are mostly interested in the LBCONN_MEM component which we will now briefly introduce. The component is on one side connected to LOCAL_BUS component and on the other provides the final module the generics and signals in Tables 3.1 and 3.2.

Table 3.1: LBCONN_MEM component connections - Generics

Name	**Usage**
BASE	The base address in the address space of the card. This address is byte oriented.
ADDR_WIDTH	Width of address bus in the user interface. This address is not in bytes but in 16b words.
USE_HIGH_LOGIC	False if the architecture supports tri-state buses, true otherwise.

Table 3.2: LBCONN_MEM component connections - Signals

Name	**IO**	**Usage**
DATA_OUT	OUT	16b data from PCI. Similar to data written to memory.
DATA_IN	IN	16b data to PCI. Similar to data from memory.
ADDR	OUT	Address bus of ADDR_WIDTH width. This address is not byte oriented, it is in 16b words.
RW	OUT	Determines whether the transaction is Write (1) or Read (0). It is therefore a Write/Read_N signal.
EN	OUT	Enable signal, active (1) when address is.
SEL	OUT	Select signal, active (1) for the whole duration of the transaction.
DRDY	IN	Data ready, when reading from module, it is active (1) when data are valid.
ARDY	IN	Address ready. Not implemented in the current version.

Here we focus on signals EN, SEL and DRDY. If we write into the module, then SEL and EN are equal. However, if we read from the module, then EN is active while there is a valid data request on address bus, SEL is active until the last data have been read from the module. DRDY determines whether the data on DATA_IN port are valid. The whole protocol is on Figure 3.4.

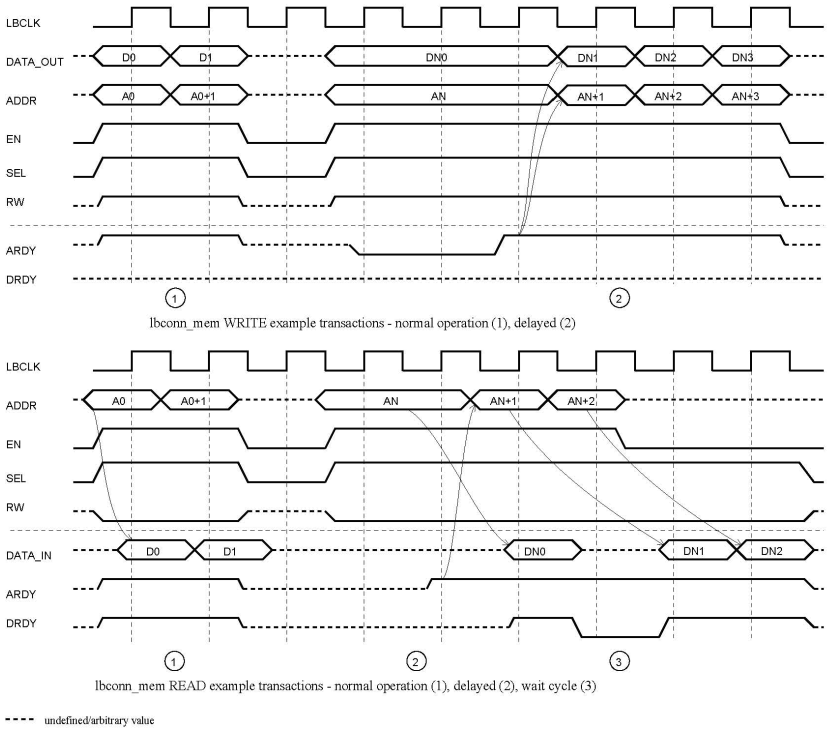

Figure 3.4: Signals during write and read operations, taken from [16]

Each 32b (dword) transaction over the PCI bus initiates two 16b (word) transactions on LocalBus, where the lower word is on an even address and the higher word is on an odd address.

For example, if our module has a base address BASE=0x3000, then writing 0x0123_4567 to address 0x3010 over PCI will initiate first transfer of 0x4567 to address 0x20 and then transfer of 0x0123 to address 0x21.

```
Regs_P : process(LBCLK, RESET)
variable addrReg : integer;
begin
   if RESET = '1' then
      into_LB <= (others=>'0');
      DRDY <= '0';
   elsif rising_edge(LBCLK) then
      addrReg := conv_integer(ADDR);
      if RW = '1' and EN = '1' then
         if ADDR(0) = '1' then
            regs(addrReg) <= from_LB xor X"FFFF";
         else
            regs(addrReg) <= from_LB;
         end if;
      end if;
      into_LB <= regs(addrReg);
      -- data are valid all the time
      -- (ie. whenever I am asked for it)
   DRDY <= EN and not RW;
   end if;
end process Regs_P;
```

Figure 3.5: Code of memory connected to LBCONN_MEM inverting top 16b of each written 32b word

Code snippet on Figure 3.5 shows a memory connected to LBCONN_MEM that always inverts top 16b of each written 32b word. Memory is represented by an array of 16b word `regs`. Signals `into_LB` and `from_LB` are connected to DATA_IN and DATA_OUT ports respectively.

4 Analysis

In this chapter we will describe the top level design, connecting the coprocessor with the outside world. Then we will the coprocessor itself, compare with the previous work and determine requirements on each part. The implementation details will be given in Chapter 5.

4.1 Top level design

Our coprocessor consists of two basic parts, a programmable microcontroller and data part with the functional units. For communication with the outside world we therefore need three basic types of communication:

- Status information – typically single 32b word containing information about status and settings of the coprocessor. Base addresses of data parts, ready/start commands, key-length, polynomial vs. normal basis etc.
- m-bit vectors – Data for the coprocessor to operate on, e.g. points to multiply, k to multiply with, curve parameters and so on.
- Microprogram – program, a set of 32b words, for the microcontroller.

We have decided to split this communication into two distinct address spaces. The first address space is dedicated for operating the coprocessor, i.e. the status information and vector transfer. The second address space is then used for reprogramming the coprocessor.

Reasoning is that while we need the status information and vector transfers for each operation, we typically do not want to change the programming very often. It is therefore useful to have this in a separate address space that can be, for example, locked from normal users.

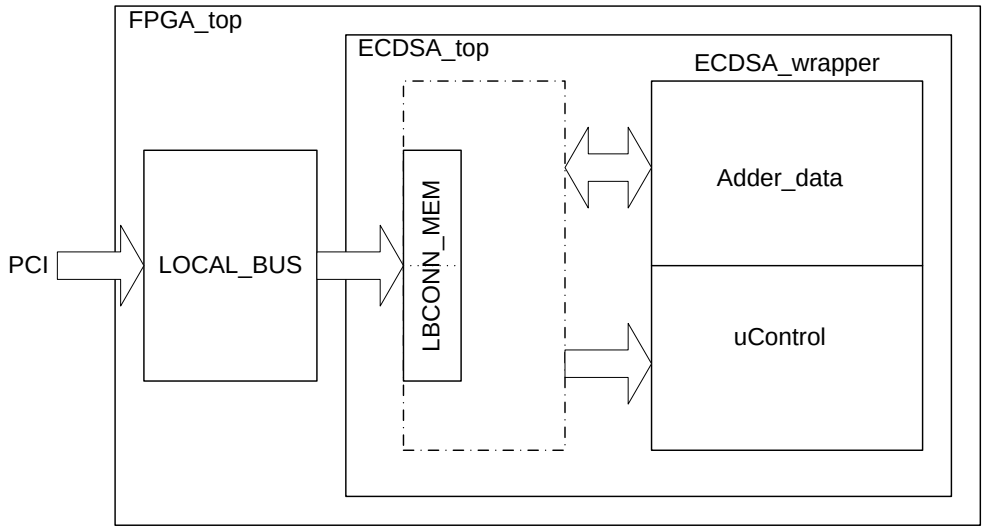

Figure 4.1: Top level design of coprocessor in Combo6X

On Figure 4.1 we see the basic scheme of the design. The FPGA_top entity is the very top level design, containing all components required for the design work on Combo6X card. Besides the modules depicted here, the design also contains several other modules, mainly design version identification and module used to test whether the device has been configured correctly. These have no impact whatsoever on our coprocessor and shall not be discussed here. We just note that the area of these extra modules has not been considered for coprocessor area evaluation.

Let us now focus on entity named ECDSA_wrapper. This entity contains the coprocessor itself, the microcontroller and the data paths. As the coprocessor core is designed to run on a frequency independent of the communication frequency, this entity also contains registers for demetastabilizing control signals when these cross between clock domains. These registers are not shown on Figure 4.1. This ECDSA_wrapper is what we will call the coprocessor, as it is virtually independent unit that could be easily plugged into any design, not just the Combo6X architecture.

ECDSA_top entity provides interface between the top level design and coprocessor. It has two main functions:

- Provide LBCONN_MEM for each base address
- Convert 16b transfers on LocalBus to m-bit vectors required by the coprocessor and vice versa

Besides the two main functions, it also provides access to configuration registers, performance counter and status registers. These will be described in detail in Chapter 5.

4.2 Coprocessor design

As we have already stated, the coprocessor consists of two main parts. The first is a data path offering all the required data operations and the second is a microcontroller that controls the data path so it does desired computations. We will first focus our attention at the data path of the coprocessor and then consider what we require from our microcontroller to be able to control the data path effectively.

4.2.1 Data path

Even through we have redone the design from the scratch, we first focus on the Figure 4.2 showing the data path architecture used in [2].

Let us now briefly discuss each component of the data path:

- Input – this is an m-bit vector coming from the outside interface.
- Output – this is an m-bit vector for the outside interface.
- Data memory – is a register file consisting of 16 or 32 m-bit registers. The number of registers depends on the desired application, 16 is minimum required for projective coordinates, 32 allows more complicated microprograms to be implemented without changing the design.

- W - is a work register, m-bit wide.
- Invertor [sic] – is an arithmetic unit fulfilling a dual role in the design. In both bases it performs field element multiplication. In normal basis it also performs element inversion, while in polynomial basis it provides field element division.
- Squarer – is a dedicated squaring unit, because in both bases squaring can be done much more effectively than multiplying a field element by itself.
- Shift – is a rotate left combination logic and is used mainly to move through individual bits of the k we multiply our point with.
- XOR – performs field element addition, which is a simple bitwise XOR.

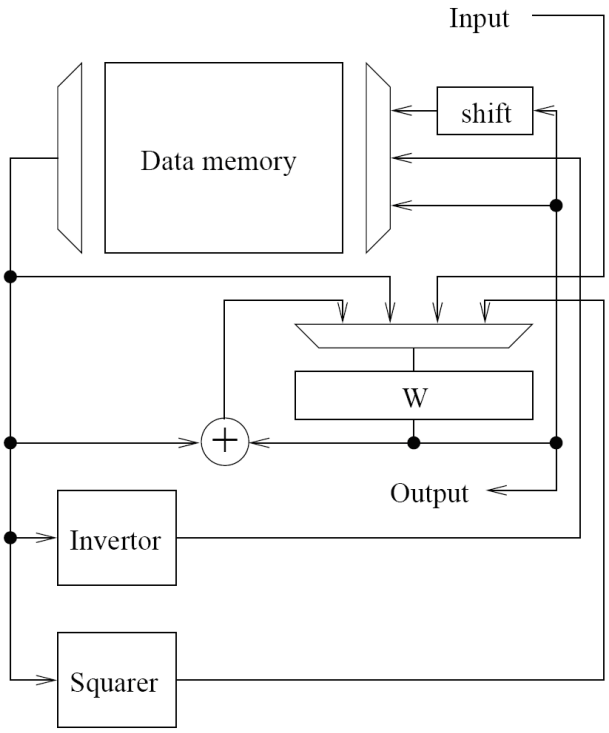

Figure 4.2: Data path architecture used in [2], figure taken from there

The design of this unit has been driven by requirement of affine coordinate algorithms for scalar multiplication (Figure 2.2, Figure 2.3). The main constrain put on the design is that, with exception of XOR, all units are fed by data from the register file. We have considered whether this would prove to be an obstacle for implementation of projective coordinate's algorithms. We found that while the algorithms as given on Figure 2.4 and Figure 2.5 could suffer a performance drop, especially where we do two squaring of an element in a row, it is very easy to avoid this by reshuffling the operations a little.

Another thing worth considering in the design are the multiplexers selecting W and memory source. While there are four different sources drawn for W register, we have to taken into account the fifth, implied, source. This source is the W itself, with function of

Clock Enable. We therefore have three sources for memory and five sources for W register. While this does not hinder performance, it is inefficient in terms of microinstruction encoding. We need two bits for memory source and three bits for W source.

We have decided that the Input wires can be easily routed to memory multiplexer. The reasoning is that the only time when we want to load data into the coprocessor is when the scalar multiplication starts. And at this time we want to load several vectors into memory and therefore have no need to load the vectors into W register first.

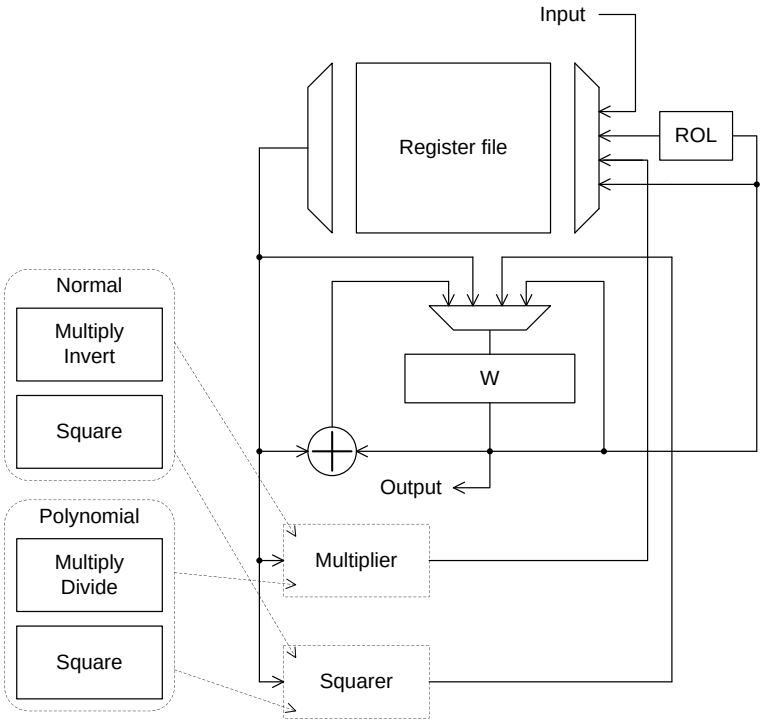

Figure 4.3: Modified data path architecture used in our design

The final architecture of data path we use (Figure 4.3) is very similar to the one used in [2]. The only difference in the design is the aforementioned rerouting of the input wire from W register multiplexer to memory multiplexer.

The rest of the changes is done for the sake of clarity of the figure. We added a wire acting as Clock Enable on W register, renamed Invertor [sic] to more fitting Multiplier and hinted that the Multiplier and Squarer units are interchangeable.

4.2.2 Microcontroller

The data path is designed to be able to perform scalar multiplication algorithm in both affine and projective coordinates. These algorithms however significantly differ, so we also need an easy-to-reconfigure controller to perform the required algorithms. We have decided to use a microprogrammable microcontroller.

First let us define the difference between a program in assembler and microprogram in microassembler (uASM). A microinstruction in uASM directly determines value on each control signal in the controller in any combination desired. An assembler instruction con-

sists of one or more microinstructions. Instruction is usually encoded in fewer bits than there are control signals in the controller, while microinstruction has one bit for each control signal. As a result, microinstructions give the programmer better control over the inner workings of the controller, but the programming itself is more demanding, while instructions require more logic inside the controller (instruction decoder) and hardwired instruction decoder, but give programmers more comfort.

Table 4.1: Control signals in data path

Name	**Width**	**Usage**
DATA_RST	1	Resets registers and finite state machines in the data path
START_MUL	1	Starts multiplication
START_DIV	1	Starts division (polynomial basis) or inversion (normal basis)
WROP	1	Write operand into Multiplier
ADDR_RD	5	Read address for register file
ADDR_WR	5	Write address for register file
MEM_WE	1	Register file write enable
W_SOURCE	2	Select signal for multiplexer before W register
M_SOURCE	2	Select signal for multiplexer before register file

Table 4.1 summarizes control signals required for our data path design. We see that there are only nine control signals a programmer has to control, so microinstructions are a viable option. Please note that while the address and select signals use more wires, there can be only one value on them per instruction, so the programmer does not have to control each wire individually.

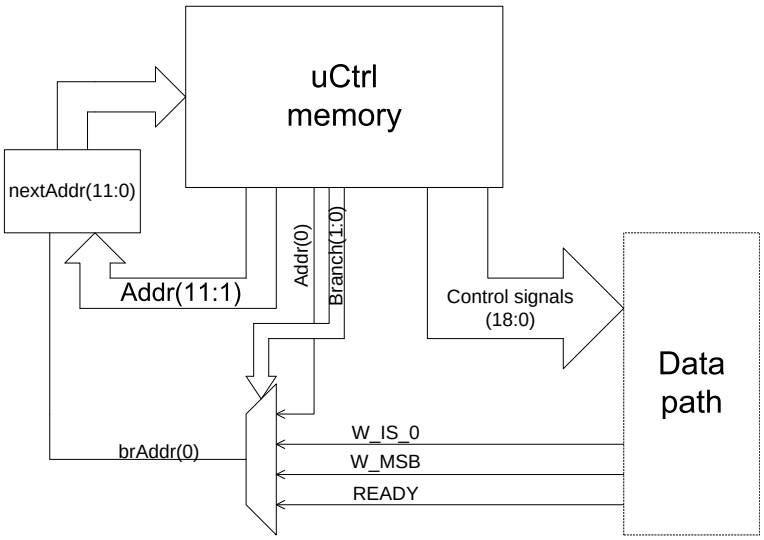

Figure 4.4: Microcontroller block diagram

Besides the control signals, each microinstruction also contains address of the next instruction and a branch condition. This branching mechanism is quite different from normal assembler programming. There we typically have a program counter register that is incremented each cycle by a constant and program goes through instructions in the order they are written in memory. To change the next address we use jump instructions, which overwrite the program counter by address of a target instruction. Conditional jumps first check their condition and then either do or do not overwrite the program counter value.

Compared to that, each microinstruction carries address of the following instruction. It is therefore possible to have the whole program in memory backwards. From the performance point of view, there will be no difference between different ordering of microinstructions in memory. The only exception is the first microinstruction, which should be at address 0, as this is the value to which address latch is reset.

We can see that there is no need for special unconditional jump instructions, as each microinstruction basically is also an unconditional jump. As we can see on Figure 4.4, conditional jump is performed by replacing the lowest bit or bits by external signals. We will explain the jumps using examples shown in Figure 4.5. The first line states, that the address of the next instruction is 01110. In the second line we see how a one bit condition is used. On the second line, the last bit of the target address is replaced by a signal and the final target address can therefore be 01110 or 01111. Unlike normal assembler instructions, microassembler also allows jumps with multiple targets. This is shown on the last line, where the last two bits of target address are replaced by a two-bit signal.

Please note that even through the target address is 01110, the four possible outcomes are not four addresses from 01110 on as we might have expected, but rather addresses 01100, 01101, 01110 and 01111. To avoid confusion in this matter, target address of a conditional jump microinstruction is always divisible by the number of possible targets, i.e. ends with at least as many zeros as there are bits of address that will be replaced.

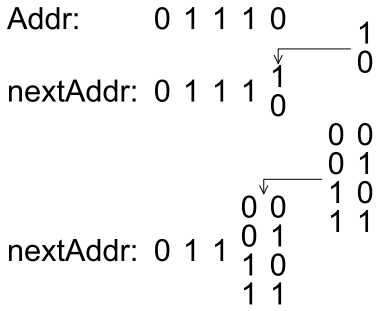

Figure 4.5: Examples of conditional jumps

For our algorithms we need only three different conditional jumps, which are summarized in Table 4.2. We also need one more branch condition which replaces the last bit of an address by the last bit of the address itself, i.e. performing the unconditional jump. This is necessary because of the way jumps are wired in the design, as is clear from Figure 4.4.

We give a full description of our microassembler in Chapter 5.4.

Table 4.2: Jump conditions

Name	**Signal is 1 when:**
IS0	Content of W register is equal 0.
RDY	Multiplier is signaling ready, i.e. the operation has finished.
MSB	The most significant bit of W is 1.

4.3 Squarer and Multiplier

First of all we should note that both Squarer and Multiplier have been provided from previous work. Considering they had been thoroughly tested already, the requirement was to use them with as minimal changes as possible. For more details on their inner workings please consult [8], [11] and [12]. In this work we shall only briefly describe the algorithms used in them.

We will first give a brief description of Squarers for each basis and then focus on the Multipliers.

4.3.1 Squarers

Squaring an element can be achieved using purely combination circuits, both in normal and polynomial basis.

For normal basis, the squaring is represented by a simple rotate left by one bit:

$$\begin{aligned} \alpha &= (\alpha_{m-1} \cdot \alpha_{m-2} \dots \alpha_1 \cdot \alpha_0) \\ \alpha^2 &= (\alpha_{m-2} \dots \alpha_1 \cdot \alpha_0 \cdot \alpha_{m-1}) \end{aligned} \tag{4.1}$$

Squaring in polynomial basis is a little more complicated:

$$\begin{aligned} \alpha &= \left(\alpha_{m-1} t^{m-1} \cdot \alpha_{m-2} t^{m-2} \dots \alpha_1 t^1 \cdot \alpha_0 t^0\right) \\ \alpha &= \left(\alpha_{m-1} t^{2(m-1)} \cdot \alpha_{m-2} t^{2(m-2)} \dots \alpha_1 t^{2 \cdot 1} \cdot \alpha_0 t^{2 \cdot 0}\right) \end{aligned} \tag{4.2}$$

We notice that the coefficients at odd positions (t has an odd exponent) are zero. Therefore it is enough to spread the squared polynomial and then just apply polynomial reduction. As this reduction also appears in the Multiplier unit, we will now give a simple example how this works. Let us say we want to square a field element represented as 1101. The irreducible polynomial we use is $t^4 + t + 1$, therefore $t^4 = t + 1$.

We will now describe individual steps of the process show on Figure 4.6:

1. This is the "spread" step, the actual squaring. We take our original $t^3 + t^2 + 1$ term and interleave it with zeros to get $t^6 + t^4 + 1$. However we need to have maximal exponent less than or equal 3. We therefore have to reduce the result.

2. We reduce the t^6 term using equation $t^6 = t^3 + t^2$, which we get by multiplying equation $t^4 = t + 1$ by t^2.

3. There is no t^5 term, so we do not reduce by $t^5 = t^2 + t$.

4. We reduce the term t^4 using equation $t^4 = t + 1$ to get the final result $t^3 + t^2 + t$.

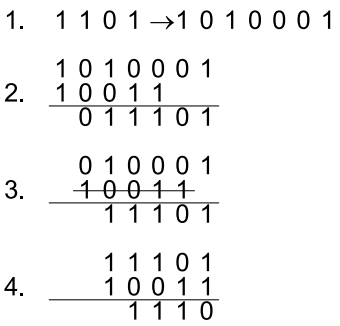

Figure 4.6: Squaring of 1101 in polynomial basis

We notice that by using the reducing polynomial from left to right, the result of reduction can never have a new 1 to the left of reducing polynomials current position. This way we can reduce a polynomial of an arbitrary length l to field polynomial length m in only $l - m$ steps.

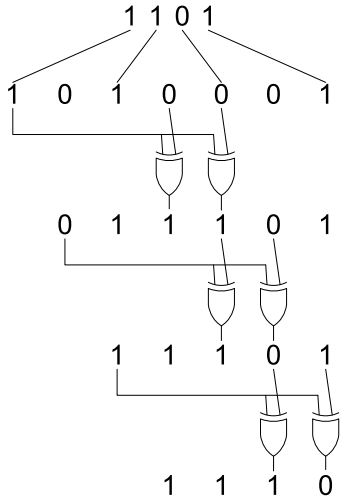

Figure 4.7: XOR network for squaring in $GF(2^4)$, polynomial basis

While this might seem like a sequential algorithm, we notice that the same result can be achieved by a simple XOR network show on Figure 4.7. This network, referred to as squaring matrix in [4], is the key to our squarer. It might seem that the network will have an unacceptable depth of logic, with one level for each reduced bit. However, it should be noted that for the sake of explanation, the Figure 4.7 shows the network in relation to the reduction algorithm, not in its optimized form. We can, for example, leave out the second row of XORs, because there will always be zero at the t^5 position. The same is true for larger vectors. We can leave out all XOR levels corresponding to reduction of odd bits. After full optimization, we get no more than three layers of 2-3 input XORs for $m = 162$.

The previous work on this topic required an external tool to generate this XOR network, which is contrary to our goal to have the design as self-contained as possible. Therefore, we have decided to leave the network generation and optimization to synthesis tools, providing only behavioral description of the reduction algorithm. Please consult the Chapter 5.2.3 for more details.

4.3.2 Normal basis multiplier

As was said before, the normal basis Multiplier unit (offering element multiplication and inversion) has been provided from previous work. This unit implements Itoh-Teechai-Tsujii algorithm [17] for field element inversion. Multiplication is done using a pipelined digit-serial Massey-Omura algorithm [6]. This multiplier can process several bits, called a *digit*, in one cycle. The only drawback of the units we use is that the number of bits processed in one cycle (*digit-width*, denoted D) has to divide the key length m. However, other versions of these units exist that do not have this constraint. For more details please see [8], [11] and [12].

There is one modification we had to do to allow key length change via a single configuration constant. To explain that, we have to briefly describe the multiplication algorithm used. The algorithm is show on Figure 4.8. Please note that the vectors have their least significant bit on the left, so *LeftShift* operation is what we usually call right shift.

Input: the multiplication matrix M for the field $GF(2^m)$; field elements $a = (a_0\ a_1\ \ldots\ a_{m-1})$ and $b = (b_0\ b_1\ \ldots\ b_{m-1})$.

Output: the product $c = (c_0\ c_1\ \ldots\ c_{m-1})$ of a and b.

1. Set $x \leftarrow a$.
2. Set $y \leftarrow b$.
3. For k from 0 to $m - 1$ do
   3.1 Compute via matrix multiplication

$$c_k := x\, M\, y^{\mathrm{tr}}$$

   where y^{tr} denotes the matrix transpose of the vector y.
   3.2 Set $x \leftarrow$ `LeftShift`(x) and $y \leftarrow$ `LeftShift`(y), where `LeftShift` denotes the circular left shift operation.
4. Output $c = (c_0\ c_1\ \ldots\ c_{m-1})$.

Figure 4.8: Normal basis multiplication, taken from [4]

The important part of the algorithm is the multiplication matrix M. This is a relatively sparse matrix. The number of 1s in a row determines the type of the matrix. Matrices with type 1 and 2 can be easily computed. Matrices with higher type require high precision mathematic libraries to compute. Please refer to [4] for more details. The requirement of high precision libraries prevents us from computing the matrices directly during VHDL synthesis. Previously, a special package containing the matrix has been generated for the required key length. However, this requires us to use external tools to generate such package every time we want to change key length. This goes against the philosophy of a self-contained design.

We therefore wanted to pre-generate multiplying matrices for all possibly key lengths.

Generating a single VHDL file containing an array of matrices proved to be unviable, because the resulting file had over 8MB and took very long to compile. We therefore decided to use ability of modern synthesis tools to load constants from a file, generated a file containing matrix representation for each key length and use an impure function to load this file as constant.

We represent this sparse matrix by an array of m by T integers, where m is the key length and T is the matrix type. The integer denotes column where 1s are in the matrix. If the integer is -1, then the number of 1s in this row is lower than T and this 1 does not appear in the matrix. Unfortunately, Xilinx Synthesis Tools can load only bit vectors during synthesis. We therefore store all matrices in bit vector format and convert them to integers in the load function. While this does increase size of the matrix files, it poses no other drawbacks and allows simple change of key length without compiling extremely large packages.

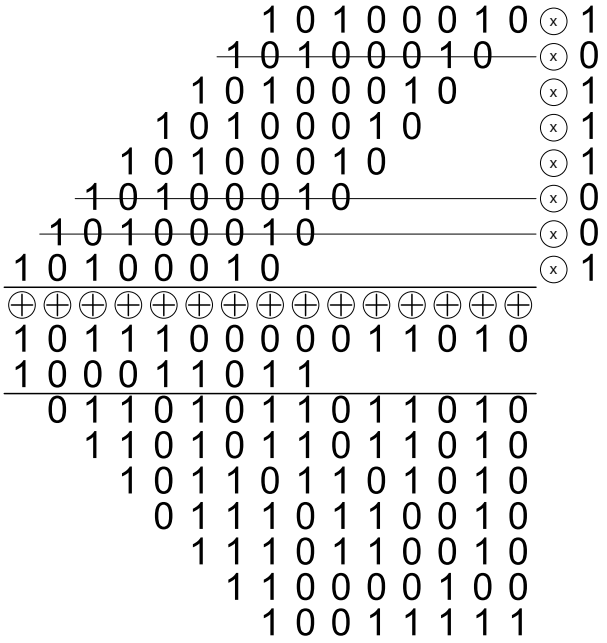

Figure 4.9: Example of multiplication in $GF(2^8)$

4.3.3 Polynomial basis multiplier – multiplication

The previously used multiplication unit did not offer any possibility of processing several bits at once. We think this is a major drawback, as it does not allow a fair comparison with the normal basis unit that supports this feature. We have therefore decided to redesign the whole Multiplier unit from scratch, implementing digit serial multiplier described in [5].

Let us first consider the multiplication itself. The example on Figure 4.9 shows multiplication of 1010010 × 10011101 in $GF(2^8)$. We take the first operand (A), multiply it by 0^{th} bit of the second operand (B). We add 2^{nd} with A shifted to the left by 1 and multiplied by 1^{st} bit of B and so on, until we have the 8^{th} line with the A shifted to the

left by 7 and multiplied by 7^{th} bit of B. We then add these eight lines together for the final sum and as the result is longer than eight bits, reduce it using the reducing polynomial $t^8 + t^4 + t^3 + t + 1 = 0$.

We notice that this is almost identical to the well-known manual decimal multiplication. There are however some differences. We do not multiply A by anything else but 0 and 1. We therefore need only m AND gates for each line, where m denotes the polynomial length. Also, considering this is multiplication in binary finite field, there is no carry between the digit places. So sum in each column can be done using only XOR gates, as odd number of 1s means 1 in the result, an even number means 0 in the result.

It should also be noted, that for the correctness of the result it does not matter whether we first add all the results together in one long $2m - 1$ register and then reduce, or immediately reduce A after shifting it left by one. See Figure 4.10.

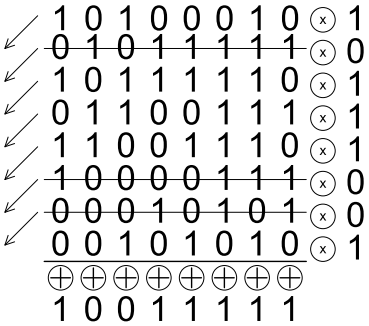

Figure 4.10: Multiplication with immediate reduction

We can see, that the number of required gates for purely combinational multiplication is m^2 AND and the same amount of XOR gates. While this might be viable for $m = 8$, we would need about fifty thousand gates for key length 162, not counting the reduction logic. It is therefore obvious that the multiplication has to be pipelined in some way.

The previous version of the multiplier used multiplication shown on Figure 4.10, computed each line in one clock cycle and accumulated in an m-bit result accumulator. After m cycles, the multiplication was finished and the result available in the accumulator.

However, we already stated we want to have a multiplier with configurable digit-width. We use a method described in [5] and depicted on Figure 4.11. It is virtually a combination of the two approaches to the multiplier. Let us describe the four steps (cycles) shown on the figure. The digit-width D is 2.

1. We multiply A by digits 0 and 1 of B, add together and store in accumulator. We shift A left by two and reduce it.

2. We multiply A by digits 2 and 3 of B, add together with the accumulator and store. We shift A left by two and reduce it.

3. We multiply A by digits 4 and 5 of B, add together with the accumulator and store. We shift A left by two and reduce it.

4. We multiply A by digits 2 and 3 of B, add together with the accumulator and store. After final reduction of the accumulator, we get the result of multiplication.

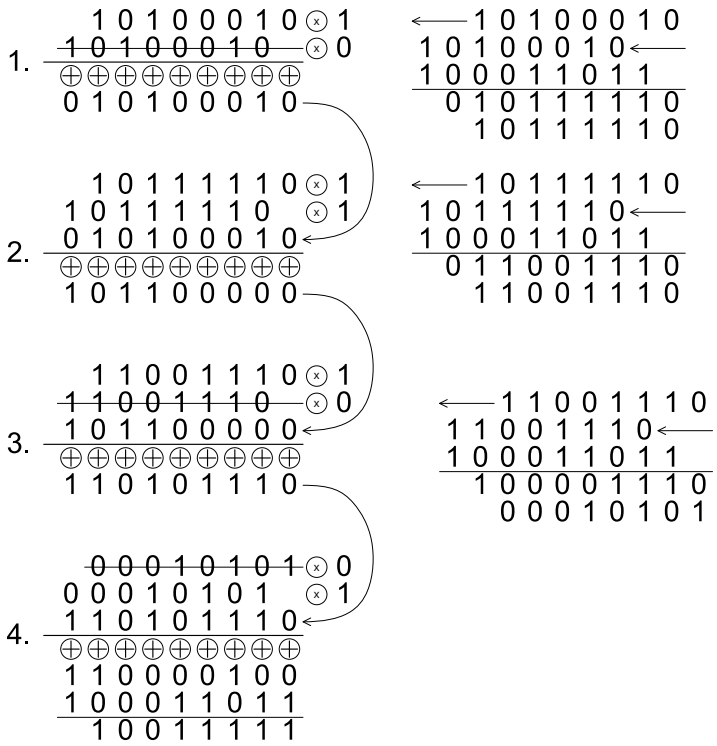

Figure 4.11: Digit-serial multiplier

The method described in [5] claims that the digit-width D should not be greater than difference between the positions of the first two nonzero coefficients in the reducing polynomial. For $GF(2^8)$, with the reducing polynomial $t^8 + t^4 + t^3 + t + 1 = 0$, the maximal D is 4. While this is advisable and we keep this recommendation, using a larger D does not make the algorithm unusable. The only difference is that the circuit that shifts left and reduces A would become a bit more complicated, as some dependencies of the reduction order are then introduced.

4.3.4 Polynomial basis multiplier – division

In normal basis, we can divide only by pairing Itoh-Teechai-Tsujii inversion algorithm with a multiplication. On the other hand, polynomial basis allows us to use Extended Euclidean Algorithm to directly compute field element division. We use the algorithm by prof. Benjamin Arazi, described in [1], found on DVD. We will now give a brief outline of how the algorithm works. Please note that we have the least significant bit as the rightmost bit of a vector, while prof. Arazi has it as the leftmost bit of a vector, which might cause some confusion.

The main difference between our approach and the approach described in [1] is that the article suggests loading register $R3$ with $0 \ldots ...01$ and computing inversion of $b(t)$, while we load it with $a(t)$ and directly compute the result of $a(t)/b(t)$. The algorithm also comes in two variants regarding the used termination condition. We will describe both variants and, as the actual difference in code is very small, we implement and compare effectiveness of both.

On Figure 4.12 we see the flowchart of the first approach. All shifts are right shifts. All additions are simple bitwise XOR of the bitvectors. *R0* is a $m+1$ bit vector, *R1*, *R2* and *R3* are m bit vectors. Operations between *R0* and other vectors use the lower m bits of *R0*. The algorithm performs the following steps:

1. Initialize – Load *R0* with reducing polynomial $p(t)$, *R1* with divider $b(t)$, *R2* with 0, *R3* with dividend $a(t)$ and *h0* with vector length m.
2. If $R1(0) = 0$ then Shift *R1* and *R3* right, goto 2
3. If $R1 = 1$ then Stop; *R3* contains $a(t)/b(t)$
4. If $R0(0) = 1$ then $R0 = R0 + R1$; $R2 = R2 + R3$
5. Shift *R0* and *R2* right; $h0 = h0 - 1$
6. If $R1(h0) = 0$ then goto 4
7. $R1 = R1 + R0$; $R3 = R3 + R2$; goto 2

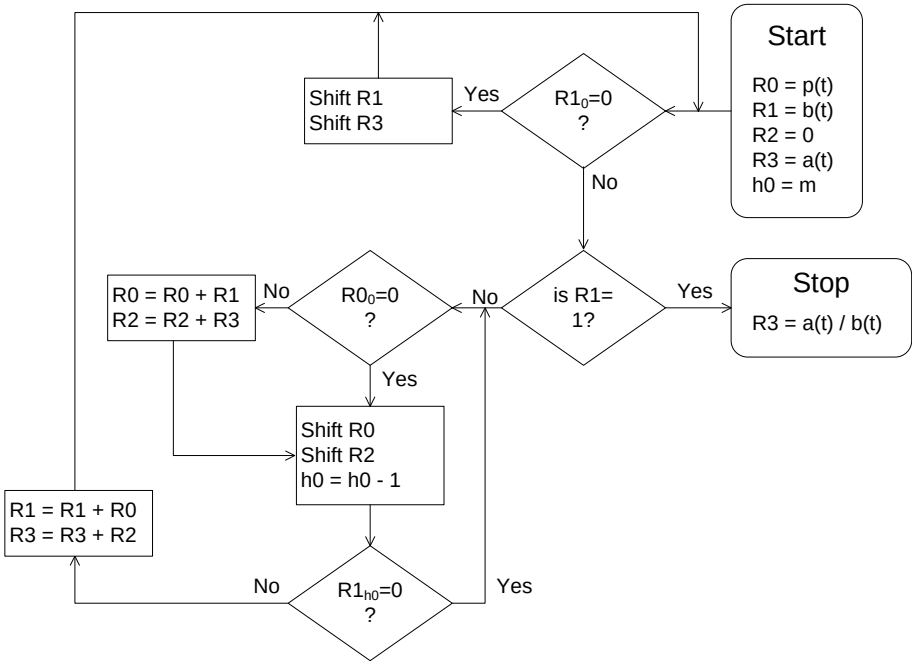

Figure 4.12: Extended Euclid Algorithm flowchart, R1(h0) variant

Checking whether an m-bit vector *R1* contains only a single 1 on the rightmost position might, for large m, require significant resources. Roughly $log_2(m)$ layers of logic containing m two-input AND gates are required to check this conditions.

Therefore, a second approach if offered in the paper, using h0 as termination condition. Flowchart of this approach is on Figure 4.13:

1. Initialize – Load *R0* with reducing polynomial $p(t)$, *R1* with divider $b(t)$, *R2* with 0, *R3* with dividend $a(t)$ and *h0* with vector length m.

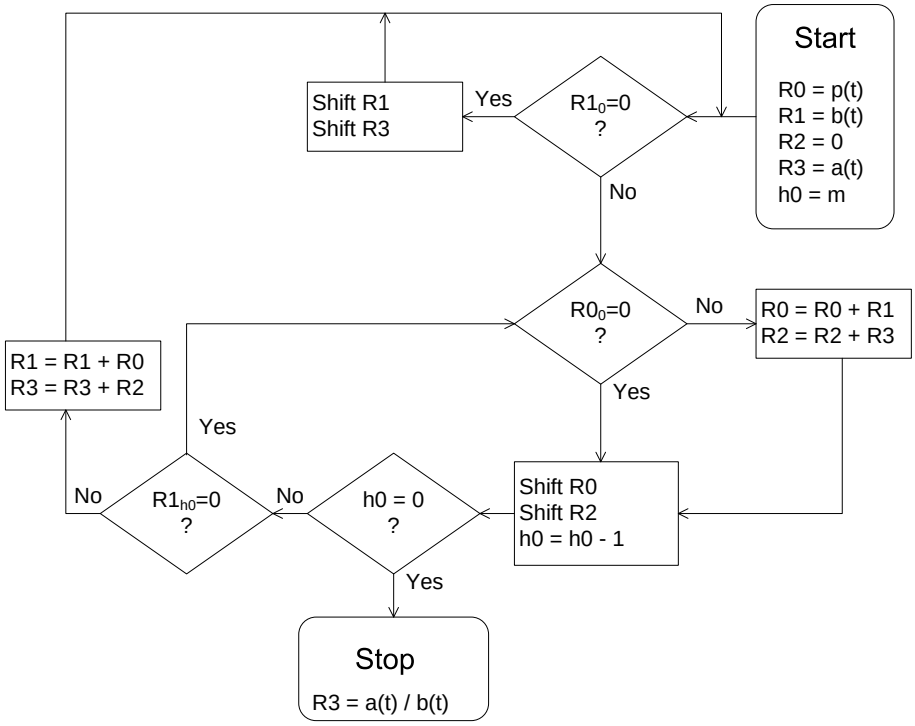

Figure 4.13: Extended Euclid Algorithm flowchart, h0 variant

2. If $R1(0) = 0$ then Shift *R1* and *R3* right, goto 2
3. If $R0(0) = 1$ then $R0 = R0 + R1$; $R2 = R2 + R3$
4. Shift *R0* and *R2* right; $h0 = h0 - 1$
5. If $h0 = 0$ then Stop; *R3* contains $a(t)/b(t)$
6. If $R1(h0) = 0$ then goto 3
7. $R1 = R1 + R0$; $R3 = R3 + R2$; goto 2

Another potentially expensive operation is extracting $h0^{th}$ bit from *R1*. We can avoid this by using an m-bit vector for *h0*, instead of an integer. We start with a single 1 in the leftmost position and then shift it right instead of decrementing *h0*. To get $h0^{th}$ bit we simply AND *h0* with *R1* and check whether the result contains a 1. While this does simplify extraction of the $h0^{th}$ bit, it might seem we lost the advantage of a simpler termination condition. However, unlike for *R1*, we know that at all times *h0* contains only a single 1 and when it reaches the rightmost position we can terminate. There is therefore no need to check the whole vector, only the lowest bit.

4.4 Verification methods

A very important part of each design project is verification and testing. Verification tests whether a design behaves according to its specification. Testing then determines whether

the actual product matches our design. It can also be said that verification is used to determine whether design does what we want and testing determines whether hardware behaves the way we told it to. In FPGA design, the testing usually focuses only on peripherals, because the chip design itself is checked by synthesis and programming tools. As we are using a well tested platform and do not implement the PCI interface ourselves, we do not need to perform testing. We therefore focus only on verification, with testing limited only to checking that the final hardware implementation gives us the same expected results.

On Figure 4.14 is an example diagram we will use to explain the basics of verification. This is not diagram of our verification setup, because, as we explain later, our setup varies from the standard model in several details. We can see several different parts:

- Testbench – the main top level entity that, for the purposes of verification, emulates the whole environment in which the design will work. Does not have any ports.
- DUT/UUT – Design under test or Unit under test. It is instantiation of the top entity of design we want to verify.
- BFM – Bus functional models of peripherals. We connect a functional model of an appropriate peripheral to design's ports. Typically, BFMs can:
  - Initiate a transfer according to the protocol.
  - Accept transfer, while checking the protocol.
  - Check whether the transfer was expected.
  - Alternatively, return content of the transfer.
  - Initiate a transfer violating the protocol in a specific manner. This serves to test DUT's ability to recover from errors
- Test – is an entity that controls all BFMs, giving them commands what to send to the device and what response to expect.
- Testcase – a specific architecture of the test entity, one for each verification test, e.g. thoroughly testing PCI and PS/2 peripherals in separate testcases.

The testbench used in our project is show on Figure 4.15. The main difference is in the PLX bus functional model. This is the bus between PCI bridge and FPGA. This BFM unfortunately does not allow an easy access to the data read from design. The test process has to directly read from the signals going between the design and BFM. Another problem is that the BFM is controlled from procedures that have to be directly in the testbench. We therefore decided abandon the system with one testbench and interchangeable test architectures and use several different testbenches, one for each test, where *tb* is the main testing process.

The standard procedure requires two separate teams. One team is doing the design and the other team is, based on exactly the same set of requirements, doing verification. Therefore, when the verification fails it can mean there is an error in design, an error in verification (testcases or BFMs) or that the requirements are too loosely specified and can be interpreted in two different ways.

Unfortunately, we could not use this scheme. We therefore faced the main danger of a single team approach, repeating the same mistakes in both design and verification. As

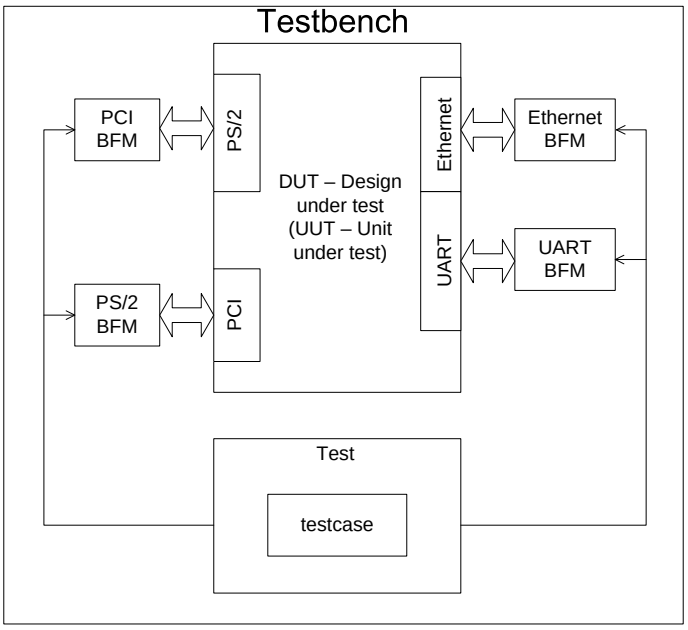

Figure 4.14: Example testbench block diagram

this was an unacceptable risk, especially on the field of cryptography, we had to choose a different approach to test the verification code itself.

The first option we considered was to use an external tool to generate inputs and expected responses. The test itself would then just load the inputs into the design and check the results against expected outputs. While this would allow us to use already tested third party tools, we would have to invoke these tools whenever we wanted to change our tests.

Another major drawback of this approach would be complicated debugging of design. If we have all algorithms implemented in VHDL, we go through them step by step and check waveforms againsts results of each suboperation.

We therefore decided to us a hybrid approach. We implemented all the algorithms in VHDL and used the external tools from [2] to test their correctness. Once the algorithms have been thoroughly tested, we use these to test the design.

We can divide the required algorithms into roughly two different groups. In the first group we have algorithms for point generation, addition, doubling and scalar multiplication in both affine and projective coordinates. These algorithms are independent on the basis used. The second group consists of base specific algorithms like field element multiplication, division, squaring. A special algorithm that should be mentioned is quadratic equation solver. This is necessary to generate points that satisfy the elliptic curve equation.

4.4.1 Code coverage

There is no objective measurement of verification quality. The final evaluation of verification is always done by a team review, where reviewers try to determine whether the verification really covered the whole design and all its options.

However, while there is no objective measurement, there are several statistics that can

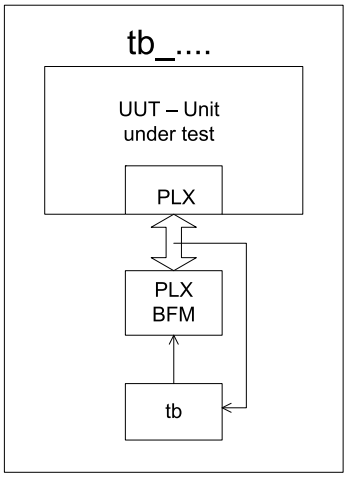

Figure 4.15: Cryptographic coprocessor testbench

help in quality evaluation. These are called code coverage and, in the basic type (statement coverage), show how many statement lines have been activated by the verification. For example statement coverage 75% means that a whole quarter of the design remains unverified, the statements were not needed during verification.

The other code coverage statistics include:

- Branch – percentage of branches taken. Each *if* can be either taken or not taken. Branch coverage 100% means that each *if* has been both taken and not taken during the verification.
- Condition – is similar to branch coverage. This however determines whether each part of each condition has been exercised. For example, condition (A=1 or B=0) that has only ever been considered when A=B=1 will have 50% condition coverage.
- Expression – is similar to condition coverage. The only difference is that while condition coverage evaluates expressions in conditional statements, expression coverage evaluates right hand expressions in assignments.
- Toggle – evaluates whether each signal and variable was toggled from 0 to 1 and from 1 to 0.

From these, the most important statistics are statement and branch coverage. We want to know that all our code has been tested and that all *if* statements were taken both ways.

There are always some statements that should not be executed during the normal course of design operation. Typical examples are assert statements in finite state machines, that check that the finite state machine does not enter an unknown state. We do not expect verification to force this situation and therefore the correct verification cannot have 100% statement coverage.

To avoid the necessity of checking whether the missing statements are all of this nature, the verification tools offer statement exclusion. In the first run of the coverage, we can manually exclude all statements that should not be included in the statistics. The following runs of the verification will exclude these statements. We can therefore expect a

well-verified design to have 100% statement coverage. Correctness of exclusion should be considered in the final review.

5 Implementation

In this chapter we will describe significant implementation details of our coprocessor. We first describe the top level entities with the main focus on ECDSA_top special registers (refer to Figure 4.1). Then we describe the data path and microcontroller and finally describe functions provided by the verification libraries.

5.1 Top level design

As was already stated, the top level entity consists of basically two parts. The first part provides communication used during the normal operation of the coprocessor, the second provides direct access to microprogram memory and allows changing the microcode. We will describe only the first part, because the second path is extremely straightforward and almost directly translates LocalBus signals into memory control signals.

Table 5.1: Control registers

Name	**Addr**	**R/W**	**Function**
readL	0x00	R	Start address of the POLY_OUT register
readH	0x04	R	End address of the POLY_OUT register
writeL	0x08	R	Start address of the POLY_IN register
writeH	0x0C	R	End address of the POLY_IN register
writeAt	0x10	R/W	*Write*: Write a polynomial from POLY_IN to register X in coprocessor register file (X is the written value) *Read*: Last written address
readFrom	0x14	R/W	*Write*: Read a polynomial from register X into POLY_OUT register (X is the written value) *Read*: Last written address
Ctrl	0x18	R/W	*Write*: bit 1: Reset; bit 0: Start microprogram *Read*: bit 1: Normal (1) or Polynomial (1) basis; bit 0: microcontroller ready; bits 16-31: key length
perfCnt	0x1C	R	Status of the performance counter in microcontroller

The communication with coprocessor is done through a set of special registers. Those are summer up in Table 5.1. The main issue with these registers comes from the mixed addressing modes in the design. Transactions over PCI bus use 32b words, so all registers should be 32b aligned. The addresses passed to tools and functions communicating with Combo6X card are byte oriented and thus all registers that give addresses have to give them in bytes. The last address mode is used by LocalBus and is in 16b words and each PCI transaction results in two consecutive LocalBus transactions. Therefore, while the *writeL* register is at address 0x08, it is at 0x04 in the LocalBus address space and refers to register number 2. Keeping this in mind, the implementation of constant registers is very

straightforward.

We will now give an example how are these registers used in the normal operation of the coprocessor. The steps are:

1. Read the *Ctrl* register, check that the basis and key length match the expected values.
2. Write 0x2 to *Ctrl* register to reset the coprocessor and performance counter.
3. Read the *writeL* register to get the base address of POLY_IN.
4. Split curve parameter A into 32b words and write them to POLY_IN register range, starting with the lowest word written to the address read from *writeL*.
5. Write 7 into the *writeAt* register. This starts process of transferring the polynomial in POLY_IN to register 7 of the register file. Our algorithm implementations expect curve parameter in register 7. See Appendix A for full list of register numbers.
6. Wait until bit 1 in *Ctrl* is 1, signaling that the coprocessor is ready for another command.
7. Repeat steps 3 to 5 until all necessary parameters have been loaded (B, k, X and Y).
8. Write 0x1 to *Ctrl*, starting the microprogram execution.
9. Wait until bit 1 in *Ctrl* is 1, signaling that the coprocessor is ready.
10. Read the *readL* register to get the base address of POLY_IN.
11. Write 18 into the *readFrom* register. This initiates transfer of polynomial in register 18 into POLY_OUT. Register 18 contains X coordinate of the result.
12. Wait until bit 1 in *Ctrl* is 1, signaling that the coprocessor is ready for another command.
13. Read the polynomial in 32b words, starting at address read from *readL*. When all is read, concatenate the result to get the desired m-bit result.
14. Repeat steps 11 to 13 to get the coordinate Y.
15. Read *perfCnt* register to determine how many clock cycles did the computation take.

From the implementation point of view, the interesting signals are *reset*, *start* and *ready*. There are two major problems with these signals. First, they have to cross clock boundary from the LocalBus clock domain to the coprocessor clock domain or vice versa. We use two flip-flops to demetastabilize the signals in each direction.

The second problem is that we want the *reset* and *start* signals to be active only until they take effect and then automatically deactivate them.

The reset is therefore set to inactive when *ready* is 1 (address register set to 0 and no transfer to or from register file in progress) and *perfCnt* is 0. The *start* signal is set to inactive is set to inactive when *ready* is 0. We assume that any microprogram loaded into the coprocessor takes longer than propagation of the *ready* signal through

demetastabilization. Should this assumption prove to be false, then the program could be run several times, depending on when does ready signal 0 propagates through. However, this would require the whole program to perform in less than three clock cycles of LocalBus clock domain. As the clock in this domain run at 100MHz, this would mean both very short microprogram and a very high coprocessor frequency.

Considering the typical usage of our implementation, we do not expect to ever encounter such combination, because scalar multiplication generally takes tens of thousands clock cycles. However, this might prove to be an issue in some future applications with extremely different frequencies. Then it would be necessary to implement some kind of hand-shake mechanism to address this issue.

5.2 Data path

The top data path entity of the coprocessor serves mostly as a framework that hosts the two main arithmetic units (Squarer and Multiplier) and the register file. Besides these elements, there are only two multiplexers, one that chooses input for register file and one that chooses input for W register, and the W register itself.

The register file is a set of 32 m-bit registers, implemented using a behavioral description. Virtex contains three different kinds of memory (for more details including coding examples please see presentation on DVD):

- Register – each bit is in its own D Flip-Flop (DFF). This is very costly (area-wise) and even though the name suggests it the right choice for registers, it is not used.
- DistributedRAM – 16 bits are stored in one Look Up Table (LUT). The ratio of LUTs and DFFs on the chip is one-to-one, so this approach requires one sixteenth of the area required by registers.
- BlockRAM – uses dedicated RAM circuits on the chips. The main problem is that for reading, BRAM required we have either the output data or the read address registered. This means that there is one clock delay between address and data.

Contrary to the general belief, register files are not implemented from DFF registers. We do not need to reset the whole register file at once, so we can use RAM instead. While BlockRAMs are the best option from the used area point of view, they cannot act as combinational logic when we read from them. This means that BlockRAMs, unlike DistributedRAM, cannot provide data in the same cycle they have been given read address. We will now explain why we require this combinational behavior.

Let us assume that we want to multiply two field elements stored in registers SX and SY. The Figure 5.1 contains two examples how such a code might look. The first example shows what we consider to be the natural way to write such code, with data request and operation on them specified in the same microinstruction.

However, this requires the following data flow:

- Cycle 0 – feed microinstruction address into microprogram memory
- Cycle 1 – register file address and control signals are available

```
   R:SX WROP ; writes SX into Multiplier
a) R:SY WROP ; writes SY into Multiplier
   MUL ; Starts the multiplication

   R:SX ; requests SX be read
   R:SY WROP ; requests SY and writes SX
b) WROP ; writes SY
   MUL ; Starts the multiplication
```

Figure 5.1: Microcode examples for multiplication: a) DistributedRAM, b) BlockRAM

- Cycle 2 – all data have to be available before this clock cycle, because the control signals are processed on the rising edge that divides clock cycles 1 and 2

This obviously requires having memory that is able to give data in the same clock cycle it has been given a read address, which limits us to using DistributedRAM. The second example on Figure 5.1 would allow us to use BlockRAMs for register file, but we consider this programming model to be very unfriendly and we therefore opted to use the DistributedRAM approach.

5.2.1 Squarers

We will now describe the Squarer unit implementation. We already said that squaring in normal basis requires only rotation left. As this is an extremely straightforward one line function, we will not describe it in any detail.

Instead, we will focus on the polynomial squarer. We stated that in the previous work, an external tool had been used to generate the required XOR network for reducing the polynomials. The tool generated a network very similar to the one on Figure 4.7. The only optimization done in this tool was leaving out the odd lines of the network, where there is always 0 in the spread polynomial. The rest of the optimizations had been left to synthesis tools.

Our idea was very simple. The synthesis tools can cope with and optimize a XOR network with the most obvious optimization (XOR with constant 0) already done. Could we perhaps use VHDL for statements to generate the XOR network in the first place? The resulting code is on the Figure 5.2. The process can be divided into two parts. In the first part we spread the input polynomial into double length. The second part then defines how to reduce the double length polynomial back into m bits. *ReductP* are the lowest m bits of the reducing polynomial. The highest bit is always 1, so we do not have to store this highest bit.

Our verification proved, that this circuit functions in exactly the same way the one from previous work does. However, we were interested not only in the functionality, but also in the optimality of the result. We compared the result for 162 bit length using Synplify Pro RTL viewer and the circuits were identical. We therefore consider it safe to assume that both approaches are equal. The main advantage of our approach is that we gained independence on external tools with no drawback at all.

```
Squarer_P : process(PolyI)
   variable Poly_int : PTypeFull;
begin
   Poly_int := (others=>'0');
   for i in M-1 downto 0 loop
      Poly_int(i*2) := PolyI(i);
   end loop;
   for i in 2*M-1 downto M loop
      if Poly_int(i) = '1' then
         Poly_int(i) := '0';
         Poly_int(i-1 downto i-M) :=
            Poly_int(i-1 downto i-M) xor ReductP;
      end if;
   end loop;
   PolyO <= Poly_int(M-1 downto 0);
end process;
```

Figure 5.2: Polynomial squarer

5.2.2 Normal basis multiplier

There were three modifications we had to do in this unit. The first two modified the entity interface to unify it with the rest of the design, the third modification concerned the way multiplication matrix was acquired.

The original entity expects data in the 0 to m-1 order, i.e. the least significant bit is on the left. Rest of the design expects this bit to be on the right. We therefore had to design a wrapper the converts between the two representations.

The second modification concerned the way the required digit-width was passed to the unit. The original design took this value directly from a configuration package. However, we wanted this value to be passed through generic, so we had to modify all entities in the unit to allow passing this generic.

The most important modification was not done on the unit itself, but on the package containing the multiplication matrix. For the reasons explained in Chapter 4.3.2 we decided to store the multiplication matrices in external files and load them as constant during synthesis. The whole process consists of three parts; the code is show on Figure 5.3.

The first part is the constant declaration itself. Please note that this has to be declared in a different package than the functions. It requires the functions to be fully defined, including their body. Function body is declared in package body, while constant is declared before that, so they cannot be both in the same package.

Next we have a function that returns filename of the file containing the desired multiplication matrix. The variable T contains the type of the desired matrix. It comes from an array generated during the matrix generation. This is array is also stored in a separate package. Value -1 denotes that we do not have multiplication matrix for the desired key length and we would be trying to open a file that does not exist which would result in synthesis error.

While this might seem to be the correct result if there is no multiplication matrix avail-

a)
```
constant ConnectTableB : TConnTableB :=
LoadConnB(FileName);
```

b)
```
function FileName
   return string is
constant c1 : string := "./nrm_data/nrm";
constant c2 : string := ".mtx";
begin
   if T = -1 then  -- When we don't have this nrm basis
      return "./nrm_data/nrm4.mtx";
   end if;
   return (c1 & integer'image(M) & c2);
end FileName;
```

c)
```
impure function LoadConnB(
    constant fname : string)
   return TConnTableB is
file mtx : text open Read_Mode is fname;
variable lineIn : line;
variable res : TConnTableB;
variable vec : std_logic_vector(15 downto 0);
variable num : integer;
begin
   if T = -1 then
      assert not NBASE
         report "Trying to use Normal Basis we don't
have (M divisible by 8 typically)"
         severity failure;
      return res;
   end if;
   for i in 0 to M-1 loop
      readline(mtx, lineIn);
      for j in 1 to T loop
         read(lineIn, vec);
         num := conv_integer(signed(vec));
         res(i)(j) := num;
      end loop;
   end loop;
   return res;
end LoadConnB;
```

Figure 5.3: Loading multiplication matrix. a) the constant declaration, b) filename of file containing the multiplication matrix, c) loading the matrix

able, we have to consider that the constant is loaded even when the design itself will use polynomial basis. Unlike normal basis, the polynomial basis is not limited in key length. Therefore, we do not want it to fail only because there is no multiplication matrix for normal basis. A default file is therefore selected.

The third function loads the matrix itself. It has to be declared as impure, because result of the function depends not only on the input parameters, but also on content of the file. It is therefore not assured that the outcome of this function will always be the same. While this does not concern us as we call it only once, it is still required by the VHDL'93 standard.

We can again see the check whether we have multiplication matrix for the given key length. This time however, there is an assert added that will make the synthesis fail if there is no multiplication matrix and the constant *NBASE* is set to true. The *NBASE* constant switches the design between normal base (true) and polynomial base (false). Therefore, if *NBASE* is true and T is -1, we are synthesizing a normal basis version of the coprocessor, but do not have the required multiplication matrix.

We can also see that we read *std_logic_vector* and the convert it to signed integer. This is because Xilinx Synthesis Tool cannot load anything but bit vectors during the synthesis, as explained in Chapter 4.3.2.

While generation of matrices of type 1 and 2 is simple, matrices of higher types require high precision libraries ([4]). We therefore used tool [7] to generate all possible matrices for key lengths between 2 and 1000.

5.2.3 Polynomial basis multiplier – data path

The polynomial multiplier consists of two main parts. A digit-serial multiplier and a divider based on Extended Euclidean Algorithm. The block diagrams of both parts are on Figure 5.4. The two big arrows show the way data are loaded into the unit. First operand is loaded and when the load signal is activated again the first operand is shifted into another register and the second operand is loaded.

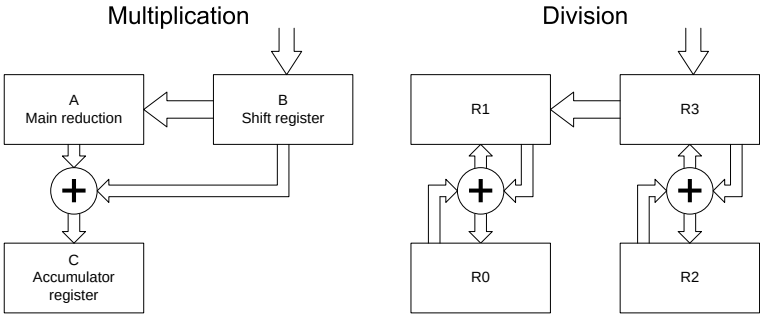

Figure 5.4: Block diagram of polynomial multiplication and division

We notice that both units use a similar set of registers. We decided to explore the possibility of implementing both algorithms over a single set of registers. Therefore, we have six different architectures to compare.

The first architecture uses *R1* as termination condition and extracts the $h0^{th}$ of *R1* by selecting it in the $R1(h0)$ fashion. The second architecture uses *h0* as the termination condition and extracts the $h0^{th}$ bit the same way. The third architecture uses *h0* as the termination condition and extracts the $h0^{th}$ bit by representing *h0* in hot one encoding, ANDing *h0* and *R1* and checking whether is any 1 in the result.

Each of the three architectures is implemented twice, once using the same set of registers for both units and once using a separate set of registers for the total of six architectures.

The reason why we are even considering using two sets of registers is that in FPGA there are usually a lot of unused flip-flops. The reasoning then is, that it might be better to build larger set of registers with relatively small amount logic connected to each than use fewer flip-flops with a lot of logic around each. The synthesis tools could place the second set of registers in the unused flip-flops with no negative impact on the chip area.

If this design is ported to ASIC, then we should always chose the single set of registers solution, because in ASIC there no unused flip-flops already wired on the chip.

To avoid the tedious task of doing six different designs, we wrote each corresponding pair of registers in a single process. This allows us to use signal declaration on Figure 5.5.

```
-- Internal A and B signals
signal A_int : PType;
signal B_int : PType;
signal C_int : PTypeExt;
signal C_red : PType;
signal R0_int   : std_logic_vector(M downto 0);

alias R1_int : std_logic_vector(M-1 downto 0) is
   A_int(M-1 downto 0);
alias R2_int : std_logic_vector(M-1 downto 0) is
   C_int(M-1 downto 0);
alias R3_int : std_logic_vector(M-1 downto 0) is
   B_int(M-1 downto 0);
```

Figure 5.5: Signal declaration for unified register set

The declaration shown will result in a single set of registers used. To switch to two sets of registers, we only have to change alias for signal in the declaration.

5.2.4 Polynomial basis multiplier – controller

Operation of the Multiplier is controlled by a Finite State Machine (FSM). As the two functions of the Multiplier are never performed at once, we can use one FSM to control both.

Control of the multiplication is fairly straightforward. There is a counter that is initially set to $\lfloor \frac{m}{D} \rfloor + 1$, where m is the key length and D is the digit-width. This counter is decreased every clock cycle. When it reaches zero it means that all m bits have been multiplied and the multiplication ends.

Control of the division algorithm is more complicated. It also differs based on the termination condition, because the two algorithms are slightly different. We will show the version that uses *h0* as the termination condition. On Figure 5.6 we show partitioning of the dataflow diagram into FSM states. Please note that while the operation blocks (i.e. the shift and adding) are drawn inside various states, they are actually on the borders between them. The operations are performed with each clock edge, which also causes a state transition.

The *ST2* state might seem to be very complicated. The dataflow suggests that the three conditions are evaluated in a serial fashion. However, please consider that in hardware, all three conditions are evaluated in parallel. Therefore the critical path through this state is only as long as the longest of the three conditions, not as a serial combination of all.

We show the FSM diagram on Figure 5.7. It is a Mealy type machine, but two states have their own defined outputs like in Moore type. The FSM is depicted this way, mainly to make the diagram more transparent. Each transition has a condition and output signals attached to them. Where in the diagram a signal is tested for equality, it is an input signal and transition condition. Where a signal is just named, it is an output signal the output value is 1. Output value of unnamed signals is 0. Transition condition *others* is used when no other transition condition is used.

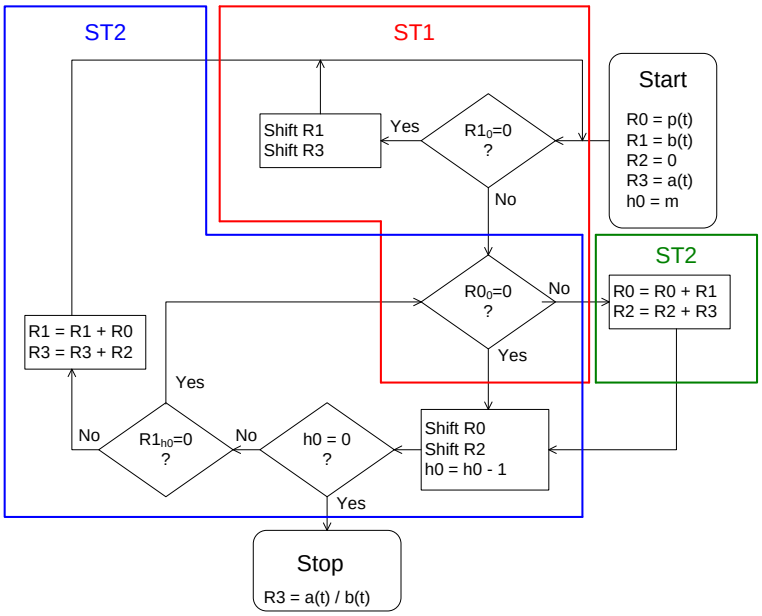

Figure 5.6: Partitioning of division into FSM states

The signals used in the diagram are described in Table 5.2. Three shifts in the table are denoted as LFSR. This means linear feedback shift register and the performed operation is shift and reduce using the reducing polynomial.

The way both left and right shift are done is on Figure 5.8. The left shift is performed on register A during multiplication. However, unlike show on the figure, the register A is shifted to the left by several (digit-width) bits. The right shift is used on registers *R2* and *R3* during division.

We do not show partitioning and FSM diagram of the division algorithm using termination condition *R1*, because they are very similar to the ones shown here.

5.3 Verification libraries

We decided to use VHDL for verification, acknowledging the risk of repeating the same errors can occur in both design and verification. We will now describe the requirements, structure and functions and procedures we implemented.

There are three basic functions we require:

- Generate a random point on a curve
- Perform scalar multiplication in affine coordinates
- Perform scalar multiplication in projective coordinates

All the functions have to be available in both polynomial and normal basis. There are two basic options how to address this requirement. The first, more direct, approach is to have an affine scalar multiplication for polynomial basis as a separate function from affine

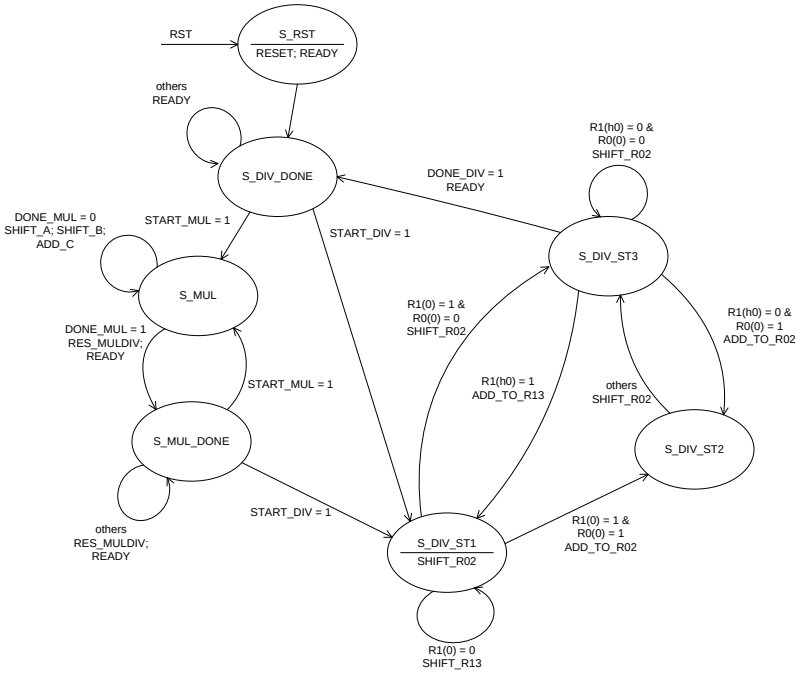

Figure 5.7: Polynomial multiplier FSM

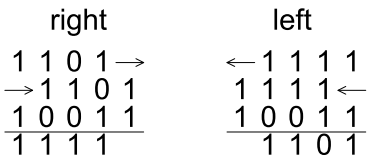

Figure 5.8: Left and right shift using reducing polynomial

scalar multiplication for normal basis. The disadvantage here is that while polynomial and normal bases have different field element multiplication, the point addition and point doubling algorithms are exactly the same. We therefore would have to code the same algorithm twice with the only difference in the names of the called arithmetic methods.

We chose another way to deal with this problem. We use a three layer structure of functions, organized this way:

- The lowest layer – basis specific functions like multiplication or division.
- The middle layer – function that allows setting the used basis and provides wrappers for all basis specific functions. The wrappers select normal or polynomial version depending on the set basis.
- The highest layer – provides the generic algorithms. The algorithms perform calls to the wrappers provided by middle layer and are therefore independent on the basis used.

Let us now focus on each of the three basic functions. The first function is random point

Table 5.2: FSM signals

Name	**I/O**	**Function**
RST	I	Resets the FSM
START_MUL	I	Start multiplication
START_DIV	I	Start division
DONE_MUL	I	Multiplication is done
R1(0	I	0^{th} bit of the *R1* register
R0(0)	I	0^{th} bit of the *R0* register
R1(h0)	I	$h0^{th}$ bit the of the *R1* register
DONE_DIV	I	Division is done
others	-	When no other condition applies
READY	O	Signals that the unit is ready for next operation
RESET	O	Resets all counters and registers in data path
SHIFT_A	O	Shift register A left by digit-width bits (left LFSR)
SHIFT_B	O	Shift register B left by digit-width bits
ADD_C	O	Add result of the multiplication to accumulator C
RES_MULDIV	O	Result source; Multiplication (1) or Division (0)
SHIFT_R13	O	Shift *R1* (normal) and *R3* (right LFSR)
SHIFT_R02	O	Shift *R0* (normal) and *R2* (right LFSR)
ADD_TO_R13	O	$R1 = R1 + R0$; $R3 = R3 + R2$
ADD_TO_R02	O	$R0 = R0 + R1$; $R2 = R2 + R3$

generation. Please note that a random point in this context means a random point on an elliptic curve, not an arbitrary combination of two random coordinates x and y.

5.3.1 Random point

First we need a function that will generate a bit vector of appropriate key length. We use the random number generator package [3] to generate eight bit random numbers and concatenate these into vectors of required length.

Using this function, we generate parameters a, b and point coordinate x. From Weierstrass equation 2.4, we get quadratic equation which we need to solve for y:

$$y^2 + xy + \left(x^3 + ax + b\right) = 0 \tag{5.1}$$

We therefore need a function that will solve a quadratic equation. Solvers for normal and polynomial basis differ, so we have to use the three layer structure to allow using the same point generation function for both bases. The solver for normal basis uses algorithm described in [4]. Solver for polynomial basis is based on algorithm implemented in [10], especially where it concerns determining whether a solution exists. It requires a square root of a field element. We implement square root by squaring the field element m-1 times (algorithm from [4]).

Both solvers require the equation to be in format $y^2 + y = c$. Please note that the constant c does not change when we transfer it from the left to the right side, because addition and subtraction are equal operations in $GF(2^m)$. We therefore use the following

procedure to transform our equation into the expected format.

$$\begin{aligned} y^2 + xy + \Big(x^3 + ax + b\Big) &= 0 \\ y^2 + a_1 y + a_0 &= 0 \\ z^2 + z = \frac{a_0}{a_1^2} &= c \\ y &= z \cdot a_1 \end{aligned} \tag{5.2}$$

We use the first two lines to transform the equation, solve the equation on the third line and use the formula on the fourth line to get the desired solution y.

5.3.2 Point multiplication

Once we have a random point, we want to perform the basic operation on it, scalar multiplication. The add-and-double algorithm itself (Figure 2.2) is the same for both affine and projective coordinates, but the point addition and doubling is coordinate specific. This time we decided to simply implement it twice, once for each coordinate system. Using the same approach we have chosen for normal and polynomial basis would bring us no advantages and more functions where a coding error can occur.

To implement the affine point addition and point doubling algorithm from Figure 2.3 we need field element multiplication, squaring and division. Squaring could be implemented using multiplication, but in both bases it can be implemented in a much faster way. Division in polynomial basis is done directly using Extended Euclidean Algorithm, in normal basis we use pair of inversion and multiplication functions.

Projective coordinates have a special algorithm for point doubling (Figure 2.4) and for point addition (Figure 2.5). Both algorithms require many multiplications and squarings. To get the final result in affine coordinates, we also need two field divisions. Another thing required are constants representing 1 and 0 in the $GF(2^m)$. While constant 0 is same in both basis and means that all bits are 0, the constant 1 is different. In polynomial basis, 1 is represented as a vector full of 0s with a single 1 in the least significant position. Normal basis represents 1 by a vector full of 1s. For convenience, we provide two functions that return these constants in the currently selected basis.

5.4 Microcode

To allow easy programming of the microcontroller, we had to develop a microassembler. Without this, programming would have to be done by directly editing individual bits of each instruction. While this is a possible approach, it is very tedious and very error-prone way to program.

We therefore developed a simple microassembler that makes programming easier and more readable for human programmers. The basic commands are summarized in Table 5.3.

The conditions that are currently supported are in Table 4.2. Addresses used in both read and write commands are named in a special configuration file (see Appendix 2.5). The names do not have any special meaning; they just make the programming more comfortable. We only need to have the register numbering consistent between the microprogram and

Table 5.3: Microassembler commands

Command	**Function**
; comment	Comments start with a colon and can be on individual line or after a command.
LB:lab1	Label Branch, target for unconditional jump. Can be followed by LC that has to be moved.
LC:lab2	Label Conditional, target for conditional jump, the following instruction will be placed to an even address. Must be followed by two non-label commands. Both following commands must contain a branch statement.
B:lab1	Branch, the next command will not be the next line but the one after the corresponding LB:lab1 statement.
BC:cond:lab2	Conditional branch, the next command will be first or second command after the corresponding LC:lab2 statement.
R:addr1	Register file read from addr1
W:addr2	Register file write to addr2
RST	Resets the data path of controller
WE	Register file write enable
MUL	Start multiplication
DIV	Start division (inversion in normal basis)
WROP	Write operand into Multiplier
WW	Source of W is W
WMEM	Source of W is register file (MEM)
WXOR	Source of W is W xor MEM
WSQR	Source of W is square of MEM
MW	Source of MEM is W
MMUL	Source of MEM is Multiplier
MROL	Source of MEM is W rotated left by one
MINP	Source of MEM is input from outside

the controlling application. Otherwise the program could write curve parameters and the multiplied point into different registers than where they are expected by the microprogram.

Conditions, register file and W register multiplexers are also named in the same configuration file so, should we desire, we could change name, order and numbering of these commands.

We give an example of normal basis multiplication (Figure 5.9, 1) and division (Figure 5.9, 2) coded in our microassembler.

The multiplication first loads both operands (*T2* and *T3*) and starts (*MUL*). Then we have to perform two dummy instructions, because the normal basis unit has its *RDY* signal invalid for two clock cycles after starting the unit. Next follows a loop that waits for *RDY* to be active and when it is, it jumps to the last branch label.

Division in normal basis has to be split into two parts. First is inversion of the divider. We load the divider (*T0*), start (*DIV*), perform two dummy operations and wait for *RDY* to be active. Please note that the inverted operand (*T0*) has to be on the units input

1)
```
R:T2 WROP
R:T3 WROP
MUL
R:T2 ;dummy for Normal unit
R:T2 ;dummy for Normal unit
B:D_MUL_RDY_1

LC:D_MUL_RDY_1
BC:RDY:D_MUL_RDY_1
W:T2 WE MMUL B:D_MUL_FIN_1

LB:D_MUL_FIN_1
```

2)
```
; X = X/Z^2
R:T0 WROP ;Invert
R:T0 DIV
R:T0 ;dummy for Normal unit
R:T0 ;dummy for Normal unit
R:T0 B:M_CONV_DINV_RDY_1
LC:M_CONV_DINV_RDY_1
R:T0 BC:RDY:M_CONV_DINV_RDY_1
R:SX WROP B:M_CONV_DINV_FIN_1
LB:M_CONV_DINV_FIN_1
MUL
R:SX ;dummy for Normal unit
R:SX ;dummy for Normal unit
B:M_CONV_DMUL_RDY_1

LC:M_CONV_DMUL_RDY_1
BC:RDY:M_CONV_DMUL_RDY_1
W:SX WE MMUL B:M_CONV_DMUL_FIN_1

LB:M_CONV_DMUL_FIN_1
```

Figure 5.9: Normal basis multiplication (1) and division (2)

throughout the whole operation, therefore we have *R:T0* on each line until the inversion finishes. Then we perform multiplication with dividend.

The affine algorithms use relatively few multiplications and divisions, so we implemented the algorithm directly using these commands. However, the polynomial algorithm requires many multiplications and the first attempt to implement it this way resulted in an unwieldy code that was very error prone and very hard to debug.

We therefore decided to include macros that are expanded into a required sequence of commands. The macros are summed in Table 5.4. The *lab* parameter is used as a prefix for labels used by the expanded commands.

Table 5.4: Macros in microassembler

Macro	**Function**
MULN(r1,r2,r3,lab)	$r1 = r2 \cdot r3$ (normal basis)
MULP(r1,r2,r3,lab)	$r1 = r2 \cdot r3$ (polynomial basis)
DIVN(r1,r2,r3,lab)	$r1 = r2/r3$ (normal basis)
DIVP(r1,r2,r3,lab)	$r1 = r2/r3$ (polynomial basis)
SQR(r1,r2)	$r1 = r2 \cdot r2$
ADD(r1,r2,r3)	$r1 = r2 \cdot r3$

Using the macros makes programming the controller much easier. However, it does not allow using direct low-level optimizations. We therefore offer access to the code after macros preprocessing. There the programmer can do the desired low-level optimizations and then proceed to compilation.

The compiler for this assembler is implemented in Java and based on a very simple split-and-compare algorithm. Each line is read, stripped of comments, split by whitespaces and each command on the line is evaluated. We decided to use the configuration file instead of hardcoded values to allow modifying the design and compilation results without actually recompiling the compiler itself.

6 Results

We evaluate the design in three aspects. The first and most important aspect is correctness of design. This is evaluated by verification. The quality of verification is estimated using code coverage. We have already stated that we do not perform any real testing, because the implementation platform and all IO units have already been thoroughly tested when the Combo6X card and the IO units were released (please consult for more information [16]).

The second aspect are synthesis results. We compare frequency and area of individual design configurations.

The third aspect is performance. We use Combo6X card to measure number of clock cycles requires to get results for a set of test points.

6.1 Verification

As we said before, the main problem of this project was that design and verification was done by the same team. We therefore chose an approach where we use a set of pregenerated vectors from [2] to test our individual functions. Once the functions pass this test, we verify the design using vectors generated and computed by them. We also compute the scalar multiplication in both affine and projective coordinates and crosscheck the results. We would like to stress that while we consider this adequate for this project, a proper verification by an independent team, including a formal review, should be performed before using this design in a commercial product.

With no formal review available for our project, we rely heavily on code coverage to prove correctness of our verification. We aim for 100% statement and branch coverage after exclusion of unreachable statements.

Table 6.1: Unit testcases

Filename	**Function**
tb_gener.vhd	Generates test vectors for performance analysis, no verification
tb_proj_base.vhd	Basic testing of projective multiplication algorithm by comparing with affine algorithm
tb_tst_af_nadd.vhd	Point addition in normal basis and affine coordinates
tb_tst_af_padd.vhd	Point addition in polynomial basis and affine coordinates, checked against values from [2]
tb_tst_norm_units.vhd	Tests normal basis operations multiplication, inversion, squaring and division against [2]

Our testcases can be divided into two major groups. The first group contains test of individual units and algorithms, the second group tests the design. The brief summary of these tests can be found in Table 6.1. We used the values from previous projects and tested polynomial point addition against them. We had no results for normal basis point addition, only basic operation tests. However, we consider this to be adequate, as the

algorithm for point addition itself has been tested by polynomial basis and therefore, if the individual operations are correct in normal basis, then the whole algorithm is.

We found one error in the design during the course of verification. When testing the projective point addition algorithm, it gave us very different results from the affine point addition one. We implemented several variants of the projective algorithm and all variants gave us the same, incorrect, result. This error was caused by a faulty function for random point generation. We generated points by choosing both coordinates x and y randomly, instead of choosing x and computing the y as described in Chapter 5.3.1.

After correcting this error, all tests pass.

Table 6.2: Scalar multipliation testcases

Filename	**Function**	**Cover.**
tb_af_norm.vhd	Normal basis, affine coordinates	100%
tb_af_poly.vhd	Polynomial basis, affine coordinates	100%
tb_pr_norm.vhd	Normal basis, projective coordinates	100%
tb_pr_poly.vhd	Polynomial basis, projective coordinates	100%

The scalar multiplication testcases are summarized in Table 6.2. All tests have 100% statement and branch coverage. To achieve this, we had to make several exclusions. We have excluded all statements that involved check for unexpected states. These are in all finite state machines, but also in all multiplexers. We also had to exclude the branch in polynomial multiplier which allows division to be immediatelly followed by another division. This situation never occurs in the tested algorithms, so the branch should never be taken.

Instead, we chose code inspection as a verification method to verify this single branch. This basically means that several reviewers read the code and discuss whether it is written correctly. A single individual test run has also been performed to check that two divisions in succession can be performed.

Because of project time constrains, we verified only for digit-width 1. This is another reason while a complete formal verification and review process should be taken before considering the design for commercial purposes. We test the correct functionality of higher digit-width in performance evaluation, where test performance and correct function for digit-width up to 20, but this does not provide code coverage and does not replace verification.

6.2 Synthesis

Once we have determined that the design is correct, there were two important aspects of design to evaluate: how fast it is and how large it is. The speed is defined by maximal frequency; definition of area depends on the chosen technology. Area in Application-specific Integrated Circuits (ASICs) is defined in square millimeters. In FPGA, we can define area by number of basic blocks used. Our implementation platform is VirtexII Pro FPGA and we therefore use the FPGA area definition.

There two basic blocks available in each FPGA:

- Slices – each slice contains (2-4 of each, depending on type):
  - D Flip-Flops implementing registers (DFF)
  - LUT4 – four input Look Up Tables that can be used for:
    - Combination logic
    - DistributedRAM
- BlockRAMs

From these we chose to evaluate the area by number of Flip-Flops, by number of LUTs used as combination logic and by the total number of slices occupied by these. The design uses BlockRAM for program memory and DistributedRAM for register file. This is same for all variants of the design and can therefore be omitted from comparison.

To compare speed, we decided to use estimation provided by Synplify Pro. The decision to use estimation instead of result of complete static timing analysis (STA) is again due time constraint on this project. When we set a requested frequency and perform the complete synthesis, the design will give us frequency very close to what we requested. If we request frequency 10MHz from a design capable of running at 100MHz, the synthesis will give us design running at approximately 11MHz. Therefore, to get the maximal frequency, we have to set a requested frequency, synthesize, set it higher, synthesize again and repeat until the synthesis fails due to timing constrains.

The problem is that each synthesis takes about 30 minutes and cannot be performed automatically. We therefore opted for estimation, which required only one synthesis per design variant. All design variants up to digit-width 20 proved to be able to run at 100MHz, because they successfully synthesized into Combo6X, where the basic operation frequency is 100MHz.

Table 6.3: Polynomial multiplier variants (m=180, D=6, V2P50)

Register sets	**Termination condition**	**DFF**	**LUT**	**Slices**	**Freq. MHz**
1	h0	1046	3501	2209	161.7
1	h0 fast	1221	3554	2238	158.3
1	R1	1045	3767	2357	161.7
2	h0	1592	3920	2418	151.3
2	h0 fast	1758	3951	2440	156.0
2	R1	1585	3957	2433	150.5

First we compare variants of the polynomial Multiplier architecture. We have three variants based on the Extended Euclid Algorithm termination criteria and each of these can operate on either one or two sets of registers. The results are in Table 6.3, we use key length 180, digit-width 6, target platform VirtexII Pro 50.

We have assumed that the second set of registers will occupy unused DFFs in slices already occupied by design. However, this is apparently not the case, as we see that the total number of slices used is higher for two sets of registers. Also, the frequency is lower in all cases, suggesting that the second set of registers required longer wire connections.

Our other assumption, that the extraction of $h0^{th}$ bit from the register R1 will be easier if *h0* is represented by a shift register instead of a counter also proved to be wrong. This variant comes out both larger and slower when one set of registers is used. In the two register set variant it still the largest design, but also the fastest choice available.

Table 6.4: Coprocessor comparison (m=180, V2P50)

	Normal			Polynomial 1 set			Polynomial 2 sets		
Digit-width	**DFF**	**Slices**	**Freq MHz**	**DFF**	**Slices**	**Freq MHz**	**DFF**	**Slices**	**Freq MHz**
1	1543	1573	214.8	1234	2022	154.6	1886	2123	154.2
2	1543	1668	214.8	1218	1963	147.2	1883	2224	156.3
3	1462	1811	214.8	1218	1977	147.2	2182	2446	170.7
4	1462	1879	214.8	1220	2099	151.4	1763	2402	169.7
5	1540	2057	214.8	1522	2372	161.1	1886	2404	156.5
6	1543	2142	214.8	1225	2244	158.3	1827	2574	171.5
9	1327	2425	214.8	1498	2605	161.1	1888	2703	158.1
12	1423	2829	214.8	1240	2617	160.0	1841	3011	166.7
15	1514	3127	211.2	1749	3172	160.2	2384	3452	171.4
20	1499	3806	214.8	1468	3158	158.4	2196	3623	167.8

The main goal of our synthesis comparison was to compare normal and polynomial basis against each other. We chose the *h0 fast* termination variant, assuming that it would offer us the best performance. However, Table 6.3 suggests this is not the case and another full set of measurements should be done.

We compared normal basis, polynomial basis with 1 set of registers and polynomial basis with 2 sets of registers. We used key-length 180 and digit-widths from range 1 to 20. Due to constrains of normal basis unit, we had to choose only digit-widths that divide key-length.

The first thing we notice is the almost constant frequency, independent on any digit-width. This is caused by the fact that the estimation does not take into account wire lengths and possible routing problems. We can therefore state only that the normal unit is probably capable of running at about 135% of polynomial unit's speed. No other conclusions can be drawn from these results.

Another think we notice is the varying number of DFFs used. The extremes deviate from the average by up to 10% for normal basis, 28% for polynomial with 1 set and 20% for polynomial with 2 sets. According to the synthesis report most of the extra DFFs were added by synthesis to lower fanout via register replication.

On Figure 6.1 we can see comparison of occupied slices in relation to digit-width used. We see that all three variants of the design are roughly equal in size. However, for lower digit-widths, we can see that normal basis with digit-width 6 occupies almost the same number of slices the polynomial basis needs for digit-width 1.

In performance comparison we will therefore focus on comparing these two digit-width sizes. However, we should bear in mind that the number of slices varies a lot and is not a perfectly monotonous function. We should therefore always consider all three variants of the design when choosing the right one for a given technology, especially when it is ASIC and not an FPGA.

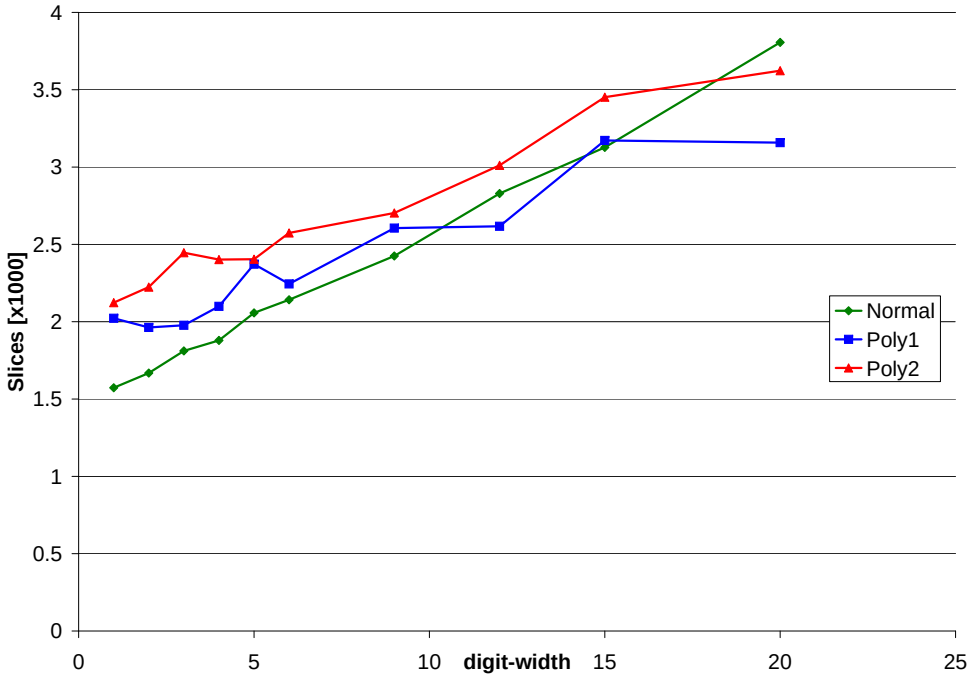

Figure 6.1: Influence of digit-width on number of slices

6.3 Performance

The last aspect in which we evaluated our design was performance. Here we compared normal basis versus polynomial basis and used both affine and projective coordinate representation. For these tests we chose hardware implementation over simulation.

We had two main reasons for this decision. The first was speed of the performance analysis. Simulation of each multiplication takes over a minute, while computing the same operation in hardware is in order of milliseconds. The second decision was to evaluate usability of Combo6X card as a cryptographic accelerator and for this we wanted to have real implementation results, not just behavioral simulation.

We wrote a simple program that can read vectors from an input file, load the into the coprocessor over PCI bus, start the computation, read back results and compare with expected values, also read from the input file.

For our testing we again chose key-length 180. We used our VHDL libraries to generate a set of thousand tests. Each of them uses a different elliptic curve, point and scalar to multiply the point with. We measure the performance by number of cycles required to finish the whole set of tests. The performance counter used counts cycles only when the microprogram is not on address 0. Therefore it does not take into account reading and writing to and from the Combo6X card. The results, with clock cycle averaged, are in Figure 6.2.

We immediately notice, that there is a big difference between normal and polynomial basis in the affine coordinate multiplication. The reason is that point addition in affine coordinates requires many divisions and those are much more expensive in normal basis,

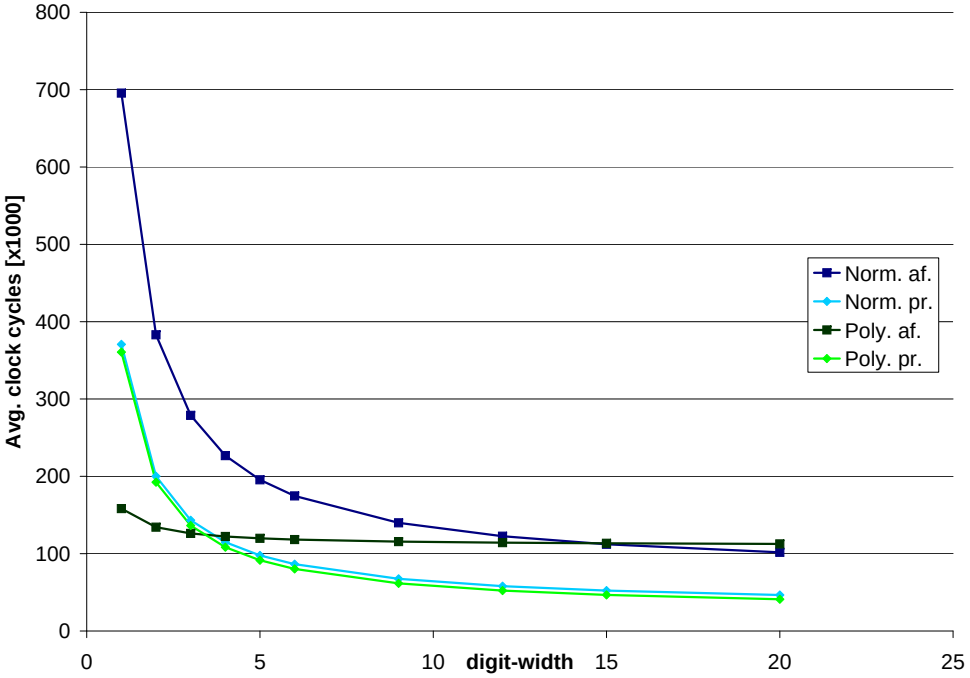

Figure 6.2: Coprocessor performance in clock cycles

where Extended Euclid Algorithm cannot be used.

We also notice that this difference drops dramatically, with normal basis even outperforming polynomial basis when digit-width reaches 2. This is caused mainly by the fact that the Itoh-Teechai-Tsujii inversion algorithm uses multiplication and therefore directly benefits from higher digit-width. In polynomial basis, the still expensive division does not benefit from higher digit-width in any way and the performance gain is therefore much lower. Affine multiplication in polynomial basis is almost invariant with regard to digit-width.

When we look at the projective coordinates, we can see that both bases have a very similar performance, with polynomial basis outperforming normal basis by a slight margin. We also notice that from digit-width 4, both bases outperform even the polynomial affine multiplication. The reason here is that both bases benefit from higher digit-width, with only two divisions in each scalar point multiplication.

Once we take into consideration the fact, that normal basis with digit-width 6 and polynomial basis with digit-width 1 are of equal size, we can say that normal basis does outperform polynomial basis, consuming only 55% of clock cycles. If we scale this by the higher frequency suggested by synthesis estimation, we get only 40% computational time with the same area requirements and both designs running at the maximal frequency. The results are summed up in Table 6.5.

Table 6.5: Average clock cycles performance

	Normal		Polynomial	
Digit-width	**affine**	**projective**	**affine**	**projective**
1	695506.31	370685.69	158257.33	360636.49
2	382975.91	200116.07	134236.24	192427.48
3	278799.11	143256.53	126229.21	136357.81
4	226710.71	114831.26	122225.69	108322.97
5	195457.67	97774.30	119823.58	91502.07
6	174622.31	86402.99	118222.18	80288.14
9	139896.71	67450.81	115553.17	61598.25
12	122533.91	57974.72	114218.66	52253.30
15	112116.23	52289.07	113417.96	46646.34
20	101698.55	46603.41	112617.26	41039.37

6.3.1 Combo6X as accelerator

We wanted to evaluate not only the performance of the coprocessor core, but also of the system as a whole, including the PCI latency etc. The basic idea was to determine whether a PCI card such as Combo6X could be used as an accelerator for computers heavily using cryptography.

We therefore measured not only clock cycles, but also the total time it took to compute the test set. The results are on Figure 6.3. We can see that the curves of total time spent on the measurement and the clock cycle curves behave in exactly the same way with no apparent performance cap caused by PCI bus latencies.

We also considered measuring performance in a batch test, where we would load a whole test set to an on-chip memory, process it in a batch and then read back all the results. This would remove most of bus latency, as there would be essentially only two long burst transfers. However, we consider this pattern to be very unlikely in normal practice and the measurement would give us no information regarding usability.

Table 6.6: Percentage of multiplication to total time

	Normal		Polynomial	
Digit-width	**affine**	**projective**	**affine**	**projective**
1	96.34%	95.32%	94.71%	97.31%
2	94.61%	95.66%	94.20%	95.59%
3	94.80%	90.73%	89.59%	90.01%
4	93.41%	92.09%	89.28%	92.98%
5	94.93%	92.07%	93.39%	86.08%
6	92.49%	91.05%	88.62%	90.52%
9	94.21%	89.10%	88.55%	88.12%
12	93.90%	88.24%	93.24%	78.81%
15	88.42%	86.43%	92.97%	85.59%
20	87.15%	85.04%	93.07%	84.27%

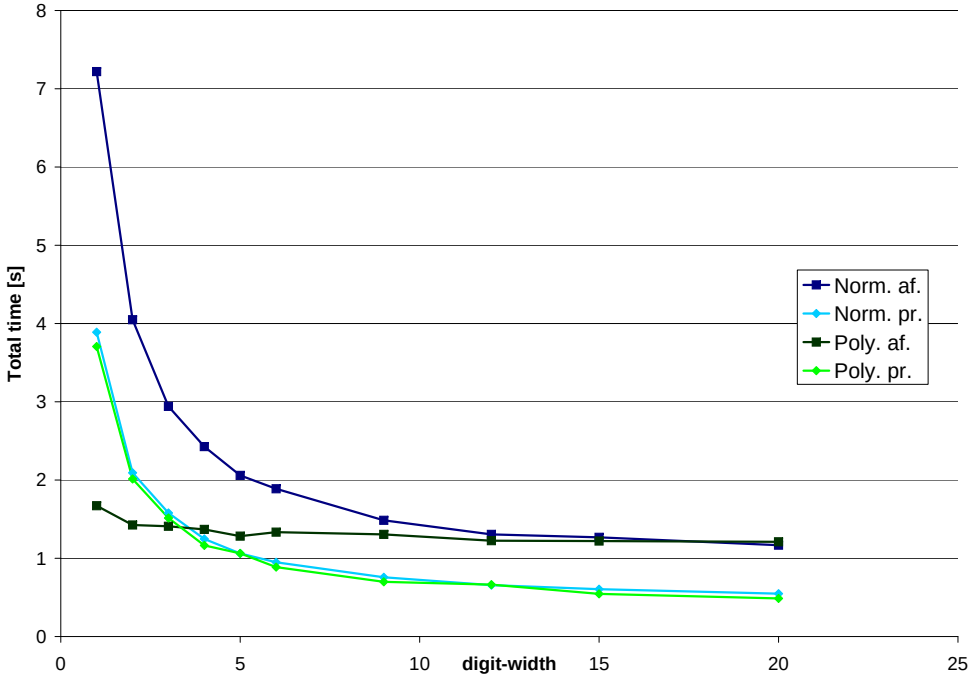

Figure 6.3: Coprocessor performance in milliseconds

Therefore, when using the card as an accelerator, we consider only individual scalar multiplications. With normal basis and digit-width 6, we get about 1ms for each multiplication. Because we do not have direct comparison with a CPU implementation, we can use only the hardware implementation to compare. We compare the time spent on the actual scalar multiplication (clock cycles multiplied by 100MHz frequency) with total time. This gives us a good insight into how much time was spent on PCI bus transfers and latencies. The results are summed up in Table 6.6. We can see that no more than 15% of time is spent on PCI bus and therefore the Combo6X card (or any PCI card in general) could be used as an elliptic curve cryptography accelerator.

However, we do not consider this to be the best way to use ECC acceleration in PC. Rather than transferring the point and curve between CPU and the accelerator, we could use the fact that the Combo6X card is mainly a network card. This allows us to consider an option of automated key exchange that would be performed only on network cards, without increasing any load on CPU.

7 Conclusions

The cryptographic coprocessor we have implemented offers a very easy way to choose the desired key-length and the basis used to represent point coordinates. Two bases are supported, normal and polynomial. Configuration of the bases is done by exchangeable arithmetic units performing multiplication, squaring and division (inversion in normal basis).

Configuration of key-length is based on a set of irreducible polynomials for key-lengths 2 to 1000, used in polynomial basis. For normal basis we developed a mechanism that reads the required multiplication matrix from external files.

Normal basis units we used supported variable digit-width for multiplication. We therefore designed polynomial units that offer the same functionality. However division is still done in single bit steps.

The coprocessor uses programmable microcontroller to control its units. Even though the microassembler we use has been developed specially for this purpose, it does offer a simple mechanism that allows its further extension. It also offers macros that are expanded into often used operations like multiplication. In this microassembler we programmed scalar multiplication for both affine and projective coordinates.

We could not use a full verification process, because that requires a separate team working solely on the verification. We tried to minimize the danger of making the same errors in design and verification by first testing our verification libraries against test vectors from previous work on this topic. The design was verified for digit-width 1 and all combinations of bases and coordinates. We focused on statement and branch coverage as the main means to determine the verification quality. However, before using the design for any commercial development a full verification and review process should be performed.

We evaluated area and speed of multiple coprocessor variants. The maximal frequency estimation proved to be unreliable and further measurements should be performed. The area also deviates slightly from our expectations (mainly the number of flip-flops, which we expected to be constant), but it is clear that area grows with increasing digit-width. We concluded that the area of normal basis with digit-width 6 and polynomial basis with digit-width 1 are about equal.

Performance testing with a wide range of digit-widths complemented verification of design and confirmed that the design gives correct results for digit-widths up to 20. We noticed a significant difference between the two bases when computing scalar multiplication using affine coordinates. The normal unit starts as significantly slower than polynomial unit, but then speeds up rapidly with growing digit-width. Digit-width has almost no effect on polynomial unit in affine coordinates. This is due to a high number of divisions and the algorithms used in each base. Both show almost identical performance in projective coordinates, with polynomial unit being only marginally better than normal basis. From digit-width 4, both units outperform the affine multiplication.

If we take into account the fact that normal unit with digit-width 6 is equal in size with polynomial unit with digit-width 1 and compare performance of the two, the normal unit outperforms the polynomial unit almost twice (55% clock cycles).

Measurements of time spent on each scalar multiplication prove that PCI bus latencies do not hinder performance of our implementation platform, Combo6X. We can therefore

use this, or a similar, card to accelerate computation of elliptic curve cryptography in PC.

This coprocessor could also be used to develop an automatic key exchange mechanism on a network card, which would allow secure communication without any increase of CPU load.

8 Bibliography

[1] Arazi B. Efficient execution of Euclid algorithm over $GF(2^n)$. Personal correspondence.

[2] Ondřej Borůvka. Kryptografický procesor. Master's thesis, CVUT, Prague, 2006.

[3] Swaminathan G. Random Number Generator Package. `http://www.janick.bergeron.com/wtb/packages/random1.vhd`.

[4] IEEE. IEEE P1363 Standard for Public-key Cryptography (Draft Version 13), November 1999.

[5] Guajardo J., Guneysu T., Kumar S.S., Paar C., and Pelzl J. Efficient Hardware Implementation of Finite Fields with Applications to Cryptography, September 2006. Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications, Volume 93, Numbers 1-3, pp. 75-118.

[6] Omura J. and Massey J. Computational Method and Apparatus for Finite Field Arithmetic. U.S. patent number 4,587,627, 1986.

[7] Schmidt J. Generator of multiplication matrix for normal basis. Personal correspondence.

[8] Schmidt J. and Novotný M. Normal Basis Multiplication and Inversion Unit for Elliptic Curve Cryptography. In *Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems*, pages 82–85, Piscataway: IEEE, 2003.

[9] RSA Laboratories. RSA-200 is factored. `http://www.rsa.com/rsalabs/node.asp?id=2879`.

[10] LiDIA Group. A C++ Library For Computational Number Theory. `http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/`.

[11] Novotný M. and Schmidt J. Two Architectures of a General Digit-Serial Normal Basis Multiplier. In *Proceedings of 9th Euromicro Conference on Digital System Design*, pages 550–553, Los Alamitos: IEEE Computer Society, 2006.

[12] Novotný M. and Schmidt J. General Digit-Serial Normal Basis Multiplier with Distributed Overlap. In *In Proceedings of 10th Euromicro Conference on Digital System Design*, pages 94–101, Los Alamitos: IEEE Computer Society, 2007.

[13] Rosing M. Implementing Elliptic Curve Cryptography, 1999. Manning.

[14] NSA. Pthe case for elliptic curve cryptography.

[15] Přibyl J. Základní matematické pojmy, popis a vlastnosti ecc. Materials for the 32UDP and 32ODI classes, CTU in Prague, FEE.

[16] Liberouter project. . `http://www.liberotuer.org`.

[17] Itoh T., Teechai O., and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverse in $GF(2^t)$ using normal bases. J. Society for electronic Communications (Japan) 44 (1986), 31-31.

[18] Zhang Y., Wei W., and Cao T. Improvement of an Authenticated Key Agreement Protocol, 2007. Springer Belin / Heiderlber.

A Register names

Number	Name	Function
0	X0	Not used
1	Y0	Not used
2	X1	Not used
3	Y1	Not used
4	X2	Not used
5	Y2	Not used
6	L	Lambda in affine point addition
7	A	Curve parameter A
8	T0	Temporary for projective addition and doubling
9	T1	Temporary for projective addition and doubling
10	T2	Temporary for projective addition and doubling
11	T3	Temporary for projective addition and doubling
12	T4	Temporary for projective addition and doubling
13	T5	Temporary for projective addition and doubling
14	ZERO	Constant 0 in a given basis
15	K	Scalar to multiply result with
16	QX	Point to multiply, coordinate X
17	QY	Point to multiply, coordinate Y
18	SX	Intermediate and result of multiplication, coordinate X
19	SY	Intermediate and result of multiplication, coordinate Y
20	SZ	Intermediate, coordinate Z
21	CNT	Counter, termintates multiplication, load with $00\ldots1$
22	ONE	Constant 1 in a given basis
23	C	Contains parameter C used in projective addition, no need to load
24	T6	Temporary for projective addition and doubling
25	T7	Temporary for projective addition and doubling
26	T8	Temporary for projective addition and doubling
27	T9	Temporary for projective addition and doubling
28	B	Curve parameter B

The naming of registers can be changed in `ECDSA.cmd` file in `uASM2` directory on DVD.

B DVD Content

```
DVD
 |-- Appendices                  - Electronic appendices
 |-- results
 |   |-- performance             - Results of performance measurements
 |   |-- reports                 - Results of digit-width comparison
 |   \-- reports_darch_180_6     - Results of polynomial unit comparisons
 |-- src
 |   |-- Compiler
 |   |   \-- uASM2               - Microassembler compiler
 |   \-- VHDL
 |       |-- compile             - Script for synthesis to Combo6X card
 |       |-- hdl                 - Design source files
 |       |   |-- normal          - Normal unit
 |       |   |-- pkgs            - Libraries
 |       |   |-- polynom         - Polynomial unit
 |       |   \-- units           - Combo6X units, property of Liberouter
 |       |-- MCSs_ver            - .mcs files loaded into Combo6X
 |       |-- nrm_data            - Normal base multiplication matrices
 |       |-- programs            - Programs used for performance testing
 |       |-- sim
 |       |   |-- models          - PLX model, property of Liberouter
 |       |   |-- tbench          - Testbenches
 |       |   \-- tests           - Random number generator library
 |       |-- tst_data            - Data used for unit tests
 |       \-- uProg               - Microprograms
 \-- thesis
```