Framework for Research of ECDSA

Tomáš Davidovič[1], Martin Novotný[1], Jan Schmidt[1],
Martin Havlan[2], Pavel Bezpalec[2]

[1]*CTU in Prague, FEE, Department of Computer Science and Engineering, Karlovo nám. 13, 121 35 Praha 2, Czech Republic*

[2]*CTU in Prague, FEE, Department of Telecommunication Engineering, Technická 2, 166 27 Praha 6, Czech Republic*

e-mail: davidt2@fel.cvut.cz, novotnym@fel.cvut.cz, schmidt@fel.cvut.cz
e-mail: havlan@fel.cvut.cz, bezpalec@fel.cvut.cz

Abstract. *Asymmetric cryptography is still very interesting field of research. Several algorithms have been proposed as a competition to the RSA standard. In this paper we will describe one of these algorithms, based on Elliptic Curves. We will briefly describe the basics of Elliptic Curve Cryptography (ECC) and its variants using polynomial and normal basis and affine and projection coordinates. Further, we will describe our hardware framework used to evaluate all possible combinations. To achieve this we implement interchangeable arithmetic units and microprogrammable controller.*

Keywords: ECC, Elliptic Curve Cryptography, VHDL, coprocessor, Combo6X, FPGA

1. Introduction

While RSA is currently the most used standard for asymmetric cryptography, there are several competing standards, offering various advantages over the RSA. One of these competing standards is Elliptic Curve Cryptography. The main advantage it offers over the RSA standard is that for the level of security ensured by 1024bit RSA keys, ECC needs only 160bit keys. It is therefore more suitable for hardware implementation, allowing the cryptographic hardware to be smaller, faster, less power-consuming or a combination of those three.

In Chapter 2 we will focus on the basic principles used in ECC, to get an insight into what is necessary to implement ECC in hardware. We will briefly describe the variants of ECC and how each variant affects the used hardware. In Chapter 3 we will describe implementation of used arithmetic units. In Chapter 4 we will describe the framework and conclude what type of IO and controller will it require and in Chapter 5 we will describe the controller and the IO unit.

2. Algorithm

The basic operation required for ECC is a scalar multiple of a point on a curve $k*P = P+P+...+P$ (k-times). For this we will need only algorithm for adding two arbitrary points on a curve.

We could do a simple algorithm that will add point P k-times to an accumulated result. However, this would take time linear to k and if we use Horner scheme (add-and-double algorithm) instead, we can do the same task in logarithmic time.

Each point on a curve is characterized by its coordinates. For ECC either affine coordinates or projective coordinates are used. In affine coordinate system, each point is represented by only two coordinates (x, y), while in the projection coordinate system three coordinates (x, y, z) represent the point.

The main difference between these two coordinate systems lies in the algorithm for point addition [4]. Algorithm for adding two points in affine coordinates (fig. 1) requires two general multiplications, one squaring and one multiplicative inver-

sion. No temporary variables are necessary. The algorithm for projective coordinates requires four squarings, eleven general multiplications, eight temporary variables, but no inversions.

Question obviously arises whether the trade-off of eight multiplications and three squarings for a single inversion is worth it. This is one of the questions our framework aims to solve.

Regardless of the coordinate system used, the coordinates are elements of $GF(2^m)$. Two bases can be used to represent field elements. The first one is a standard polynomial basis generated by an appropriate irreducible polynomial. Each coordinate of length m is represented as a polynomial:

$$\alpha_{m-1}x^{m-1} + \alpha_{m-2}x^{m-2} \ldots + \alpha_1 x + \alpha_0 1$$

Normal basis represents coordinate of length m as polynomial:

$$\alpha_{m-1}x^{2^{m-1}} + \alpha_{m-2}x^{2^{m-2}} \ldots + \alpha_1 x^2 + \alpha_0 x$$

Output: the point $P_2 := P_0 + P_1$.

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 1. Addition of two points (affine) [4]

3. Arithmetic units

The actual vectors of m bits represent coefficients α belonging to appropriate power of x. All point operations can be expressed using the following field operations: addition, multiplication, division (or inversion) and squaring [4].

Addition and subtraction are the same for both bases and are represented by a simple bitwise XOR. In Chapter 3.1 we will present fast squaring and Chapters 3.2 and 3.3 will focus on multiplication and division respectively.

3.1 Squaring

While we could perform squaring via multiplication by the same coordinate, there is a faster method for each basis. In normal basis, moving a coefficient to the left by one means it is moved to double power of x. Squaring in normal basis can therefore be represented by a simple rotation to the left.

Polynomial basis squaring is a bit more complicated. First we spread the bit representation to double length. This way each coefficient is moved from x^k to x^{2k}. We are now missing coefficients for odd powers of x. Since squaring cannot produce odd powers of x, we will simply insert zeros. The last step is reduction from the double length back to the original length. This can be done using a fixed circuit that, for small reduction polynomials (trinomials and pentanomials) has depth of two or three XOR gates. The whole squaring is therefore done by two or three layers of logic and thus is extremely fast.

3.2 Multiplication

In both coordinate systems, multiplication represents the most used operation. Therefore, it is vital for high performance to have as fast multiplier as possible.

Polynomial multiplier evaluates the expression:

$$C = A \times B \bmod F(x),$$

where $F(x)$ is the field polynomial.

The basic polynomial multiplier is a LSB (least significant bit) multiplier, also called bit-serial multiplier. Based on a bit in B, it either adds or doesn't add an appropriately shifted (and reduced) polynomial A to the current result C in accumulator. Once it went through all bits in B, the result of multiplication is ready.

The whole multiplication takes m clock cycles, where m is the length of multiplied polynomials. While this gives the fastest circuit in the means of clock frequency, it is possible that the result can be obtained in lower time using a digit-serial multiplier that multiplies by several bits at once. Another reason for using digit-serial multiplier is that critical path delay in the multiplier is not the only factor influencing the maximum possible frequency of the whole circuit. Therefore it is advantageous to

have means to tune multiplier performance to whatever arbitrary frequency is enforced by the used crystal or other, slower, parts of the circuit.

The basic operation of a digit multiplier is similar to bit-serial multiplier, only instead of multiplying by a least significant bit we multiply by a least significant digit. Assuming digit width D, we will start with the lowest D bits from polynomial B.

We then take D copies of the polynomial A (shifted by 0 to D–1 bits to the left). For each copy we decide whether it should or should not be added to the result (based on the appropriate bit in the digit). We add all D copies together; using to our advantage that addition requires just a single XOR and add the result to the result accumulator [3]. The principle is shown on Figure 2, where b_{Di+N} denotes N-th bit in the i-th lowest digit.

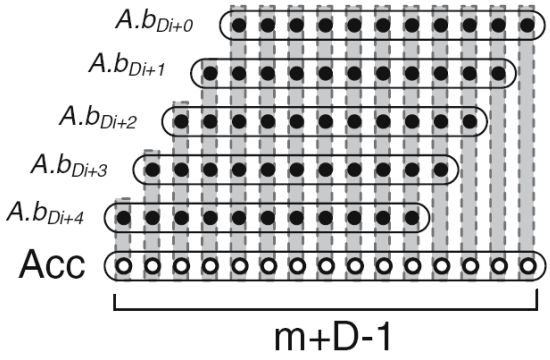

Figure 2. Multiple digit adder core [3]

For the next step we multiply the polynomial A by x^D, i.e. shift the polynomial A by D bits to the left. This means we don't need a circuit for left shift by an arbitrary number of bits but use only D different fixed-function circuit. Another D bit digit is taken from the polynomial B, filled with zeros if there are not enough bits left in B and the whole process is repeated.

At the end, the result is accumulated in the result accumulator of length $m+D-1$ bits. The actual result C is obtained by reducing content of this accumulator using field polynomial $F(x)$.

Multiplication in normal basis is computed using a pipelined bit-serial Massey-Omura [6] multiplier. It also allows simultaneous processing of D bits in a very similar manner.

3.3 Division

For affine coordinates we also need one division for each point addition. Generally, division is more expensive than multiplication in both bases.

The normal basis doesn't allow direct division, or at least not in a form that could be easily implemented in hardware. We therefore first compute an inversion of the given field element and then multiply by this inversion.

We use the Itoh, Teechai, Tsu-jii (ITT) [5] algorithm, that is the fastest known algorithm calculating the inverse element according to the Fermat's little theorem (this theorem states that $a^p=a \bmod p$, hence $a^{p-2}= a^{-1} \bmod p$). ITT uses repeated multiplications and squarings.

This principle is universal for all bases. However, for polynomial base there is a better algorithm giving us direct result of division. It is a slight modification of Extended Euclidean Algorithm (EEA) [7].

This algorithm is faster than the ITT, but requires a new circuit, besides the existing addition, squaring and multiplication. However, this special circuit uses only four registers and as we will never use multiplication and division in parallel, we can easily combine the two units to operate over the same set of registers.

This results in fewer flip-flops used than we would require for two separate circuits, but there is more complicated logic bound to each of the four registers.

Whether this tradeoff will actually pay off is dependent on a given technology. For ASIC, where we build a fully customized circuit, it is better to trade off some small amount of logic for $4 \cdot m$ flip-flops. FPGA, on the other hand, comes with a prepared structure and each cell contains a flip-flop and some small amount of logic. Praxis shows us, that common designs utilize flip-flops in only half of the used cells. We therefore assume that giving each unit a separate set of registers will, in FPGA, result in a circuit of roughly same size and possibly greater speed.

4. Framework

In the previous chapter we have concluded that while addition and subtraction is the same for both bases, both squaring

and multiplication unit are dependent on the used base.

Therefore, it is not possible to have a circuit that will work on both bases using the same arithmetic units. There are two possible solutions to this problem.

First is to design a framework with universal interface for the arithmetic units. Using a different base is then a matter of simply swapping the units in design and running the synthesis again. The disadvantage of this approach is that once it has been synthesized we have a coprocessor that can work on only one base.

The second approach deals with this problem in tradeoff for larger area. It incorporates both units in the design and allows switching between the used bases by setting an appropriate configuration register.

Regardless of the approach used, we will require some kind of register file (data memory), a work register and a shifter. The basic block diagram of the first approach is on figure 3.

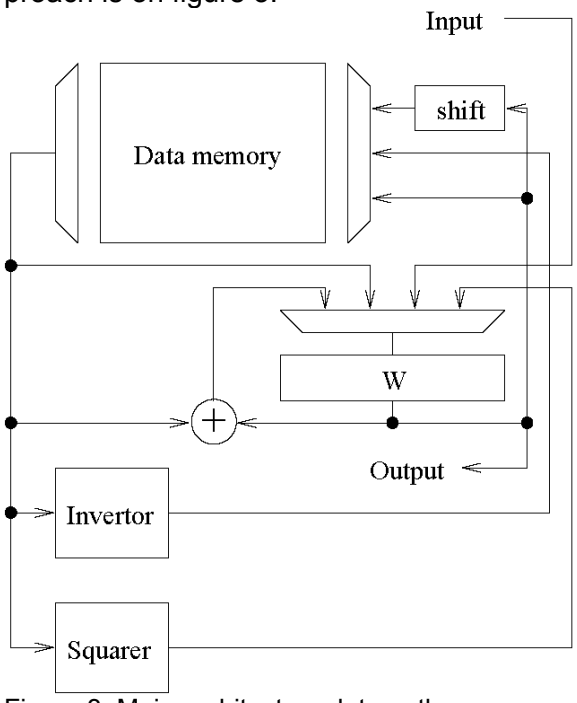

Figure 3. Main architecture data path

So far we have focused only on the data path of the coprocessor that will allow us to perform all the elementary operations required for point addition.

We will obviously also need some kind of controller that will not only direct the units to perform a single point addition, but also control the whole process of scalar point multiplication.

We will also need some means to pass data between the coprocessor and other units in the hardware. We will focus on these in Chapter 5.

5. Controller and IO

We have concluded that besides the arithmetic units, we will also need special interface to pass data between the coprocessor and other units in the hardware and we will need a controller that will direct the computation itself.

In Chapter 5.1 we will focus on various options for the controller and in Chapter 5.2 we will describe the current state of our IO unit.

5.1 Controller

Our goal is to have the controller independent on the base used. This is relatively trivial task, given a suitable interface for connecting arithmetic units. Squaring unit requires no control at all, as it is a simple combinational circuit.

Multiplication can be controlled by a pair of start/done wires. The problem arises with division/inversion unit. The polynomial base unit is more powerful than the normal base one, because it also allows direct division in addition to just inversion.

We have decided to sacrifice this ability for the sake of universality, but the advantages and disadvantages are still evaluated. The current version however allows only inversion and both units can thus be controlled by another pair of start/done wires.

While the controller can be made independent on the used base, it still does depend on the coordinate system used.

We decided to use programmable microcontroller. The main advantage of this approach is that it is very easy to change coordinate system and to fix problems by simply loading different firmware.

Using the second approach to the framework, that contains both sets of arithmetic units, we can make a circuit that will perform scalar point multiplication with any combination of base-coordinate system, based on a single configuration register.

It is obviously possible to store the firmware in ROM instead of a RAM/Flash memory. This would sacrifice the reconfigurability of the design in exchange for lower power and area consumption.

To allow easy firmware modification, we designed a very simple microassembler specially for this controller. Compiler for this assembler has two output files. One is a bitstream for firmware upload, the other is a VHDL file that specifies content of ROM, should a fixed function circuit be desired.

5.2 Input/Output unit

Designing an IO unit without a specified interface puts two contending demands on the designer. On one hand, we want the interface to be as fast as possible. On the other hand, we need the interface to be as universal as possible.

We decided to use interface with full handshake, allowing our coprocessor to operate on a frequency different from the bus frequency. The bus width is configurable during synthesis, but once the circuit is done, it is fixed to the given width.

We allow two approaches to the device. In the simple serial transfer, the circuit first expects the coordinates of the multiplied point and then the value k to multiply it with. Once all the data are loaded, the circuit starts its operation. There is no status register or "done" wire. Instead, each attempt to read from the circuit while it operates will simply not be acknowledged until the computation is done and the data are ready.

The second option we offer is more suitable for address-and-data buses, like PCI or memory interface. It offers an address space where to read and write bit vectors of the appropriate length, an address register and status register.

To perform a write, we first write the m-bit vector into the vector address space, then write desired target address into the address register and then wait until DONE flag appears in the status register.

Read operation is very similar, except that we first write the desired address, then wait for the DONE flag and then read the bit vector from its address space.

Direct access to the coprocessor memory is not allowed. The memory organization is such that each address contains a single m-bit vector. The IO bus generally will not be m-bit wide, so direct access to the memory would require a read-modify-write cycle and, combined with the handshake, would add logic directly into the core. Our approach allows transformation from bus width bit vectors to m-bit vectors outside the core.

We will evaluate the direct access option for bus operating in the same clock domain as coprocessor, which should allow us to simplify the necessary logic.

6. Future work

Adder, squarer and multiplier in their current form fulfill all requirements of the project. We will make further research into the division and inversion units. While the normal basis unit has already been made scalable (i.e. flexible in terms of number of bits processed in parallel) [8], the scalability options of the polynomial unit should be further investigated. This property might suppress the disadvantage of affine coordinates having one division in each point addition.

Further, we will focus on evaluating various configurations of circuit in both FPGA and ASIC designs, as we have a reason to believe that the best configurations for each technology will differ.

We will evaluate different versions of firmware for our controller. The main focus will be to exploit the advantage of direct division offered by polynomial unit. We believe that in combination with digit-serial divider, the speed up of the circuit would outweigh the disadvantage of special program for each base.

Concurrently with changes to the firmware, we will also modify the microassembler we use to offer a better fit into standard 32-bit wide memory. To compensate for the aforementioned problem with different code for both bases, we will research possibility of adding effective means of function calls or macros.

We will design a new IO unit, that will allow connection to bus operating in the same clock domain as the cryptographic core and we will compare it with our current asynchronous unit with handshaking.

The cryptographic coprocessor will be implemented on Combo6X card from

project Liberouter. We will connect our coprocessor to the card's current network framework using both synchronous and asynchronous IO unit and compare the impact it has on the performance.

7. Conclusions

We have described the basic principle of Elliptic Curve Cryptography and deduced the basic arithmetic operations required. We considered and described the implementation of arithmetic units performing these operations and compared the differences between units for normal and polynomial bases.

Further, we described the current state our framework used to evaluate these units and possible approaches concerning the tradeoff between circuit area and its universality. We also described reasons for using a programmable controller over a fixed function one and concluded that an universal IO unit poses many problems and it is necessary to write special IO unit for each given interface to obtain the highest possible performance.

8. Acknowledgement

This work has been supported by CESNET, project 140R1/2005.

9. References

[1] Bečvář, M. - Schmidt, J.: Reconfigurable Acceleration of Intel PC: A Quantitative Analysis, Proceedings of IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop. Gyor: Széchenyi István University of Applied Sciences, 2001, s. 93-96. ISBN 963-7175-16-4.

[2] Borůvka, O.: Kryptografický procesor, CTU in Prague, Diploma thesis

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

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

[5] Itoh, T. - Teechai, O. – Tsujii, S.: A Fast Algorithm for Computing Multiplicative Inverse in $GF(2^t)$ using normal bases. J. Society for electronic Communications (Japan) 44 (1986), 31-36.

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

[7] Arazi, B.: Efficient execution of Euclid algorithm over $GF(2^n)$; unpublished

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