Final Project Report: Cryptoprocessor for Elliptic Curve Digital Signature Algorithm (ECDSA)

Team ID: IN00000026
Team member: Kimmo Järvinen
tel. +358-9-4512429, email. kimmo.jarvinen@tkk.fi
Instructor: Prof. Jorma Skyttä
tel. +358-9-4512450, email. jorma.skytta@tkk.fi

Helsinki University of Technology, Signal Processing Laboratory
Otakaari 5A, FIN-02150, Espoo, Finland

August 7, 2007

Abstract

Elliptic Curve Digital Signature Algorithm (ECDSA) is implemented on an Altera Cyclone II EP2C20F484C7 FPGA using a DE1 development and education board. Digital signatures are digital counterparts of handwritten signatures. They provide proof of authorship and authenticity and they are unforgeable. They also provide proof that the document has not been altered after signing. The design includes a Nios II processor together with custom-designed modules for elliptic curve cryptography, SHA-1 hash function and modular arithmetic. A pseudo-random number generator is also included for rapid and secure generation of pseudo-random numbers. A user interface is designed with Nios II Integrated Development Environment (IDE) for demonstrating the use of the design. The design requires approximately 85 % of the device resources. Signature generation is computed in 0.94 ms and signature verification requires 1.61 ms.

1 Preliminaries

Research on hardware implementation of cryptographic algorithms has been intensive during the recent years. Field-programmable gate arrays (FPGAs) are very attractive platforms for implementing cryptographic algorithms for various reasons including performance, flexibility and cost efficiency [15]. This report presents an efficient implementation of Elliptic Curve Digital Signature Algorithm (ECDSA) by using a standardized curve B-163 which is listed in [11].

Digital signatures play a central role in modern cryptosystems. They can be viewed as digital counterparts for handwritten signatures and they are authentic,

unforgeable and non-reusable. A signed document is also unalterable and the signature cannot be repudiated meaning that the signer cannot afterwards claim that (s)he did not sign the document. [13]

Digital signature algorithms are public-key cryptographic algorithms and thus they involve two keys; one which is private and one which is public. A document is signed with the private key and the signature is verified with the public key. Only the private key needs to be kept in secret in order to prevent other people from forging one's signature.

Public-key cryptographic algorithms are based on mathematics (or number theory to be more precise) and it is impossible to discuss these algorithms without any math. The focus of this report is in implementing ECDSA on an FPGA and details of the algorithms are consider only to the point which is necessary for understanding the implementation. If more detailed descriptions of algorithms are wanted, then the reader should consult references which are listed in the end of the document.

The implementation is optimized especially for Altera FPGAs and it is designed to take advantage of embedded memory blocks inside Cyclone II. Most of the design efforts have been dedicated to elliptic curve operations which are the most time consuming operations in ECDSA, by far. The results show that even a low-cost FPGA such as Cyclone II can be efficiently used for implementing complex high-security public-key cryptographic algorithms.

Sec. 1.1 presents the basics of elliptic curve cryptography and Sec. 1.2 describes ECDSA. Hardware architecture is presented in Sec. 2 and a user interface implemented by using Nios II IDE is considered in Sec. 3. Results are presented in Sec. 4 together with discussion on them. Finally, the report ends with a list of possible improvements.

1.1 Elliptic Curve Cryptography

The theory of elliptic curves is deep and an enormous amount of research has been done on elliptic curve cryptography during the past twenty years or so. Therefore, it is impossible to present an extensive review of the field here and only subjects which are the most relevant are discussed in the following. Interested readers are referred to [3], for example, for further information.

All elliptic curve cryptosystems are based on an operation called elliptic curve point multiplication which is defined as

$$Q = kP \tag{1}$$

where k is an integer and Q and P are points on an elliptic curve. A point is represented with two coordinates as (x, y).

The reason why elliptic curve point multiplication is used in cryptosystem is that it is relatively easy to compute but its inverse operation called elliptic curve discrete logarithm problem, that is finding k if P and Q are known, is considered

impossible to solve with present computational resources if parameters are chosen correctly. Thus, elliptic curve discrete logarithm problem can be compared, for example, to integer factorization problem which is used in the popular RSA cryptosystems. There is, however, a notable difference because sub-exponential algorithms for solving elliptic curve discrete logarithm problem are not known and, therefore, key lengths can be shorter than in RSA.

Elliptic curve point multiplication is computed by using two principal operations; namely, point addition and point doubling. Point addition is the operation $P_3 = P_1 + P_2$ where P_i are points on an elliptic curve. Point doubling is the operation $P_3 = 2P_1$. In this design, point multiplication is computed with the so-called Montgomery's ladder [10] which operates as shown in Alg. 1 of the Appendix.

Elliptic curves used in cryptosystems are defined over finite fields denoted by $GF(q)$ where q is the number of elements in the field. It is commonly preferred especially in hardware implementations to use binary fields $GF(2^m)$ where an element of the field is presented with m bits. In this design, the field $GF(2^{163})$ is used and it is constructed by using normal basis. Arithmetic operations are computed as follows:

- Addition $a + b$ is computed with a bitwise exclusive-or (XOR).
- Multiplication $a \times b$ is computed as presented by Wang et al. in [14]. This multiplier structure is referred to as Massey-Omura multiplier and it is discussed in Sec. 2.1.1.
- Squaring a^2 is simply a cyclical rotation of the bit vector representing a.
- Finding an inverse element a^{-1} such that $a^{-1} \times a = 1$ is performed as suggested by Itoh and Tsujii in [4] and it is called henceforth Itoh-Tsujii inversion. One Itoh-Tsujii inversion requires 9 multiplications and 162 squarings if $m = 163$ [4].

Point representation with two coordinates as (x, y) is referred to as the affine coordinate representation. When points are represented in affine coordinates, both point addition and point doubling require inversion in $GF(2^m)$. Inversion is by far the most expensive operation and, thus, it is advantageous to trade inversions for multiplications. This can be done by representing points with projective coordinates as (X, Y, Z); that is, with three coordinates. Mappings between these two representations are performed as $(x, y, 1)$ and $(X/Z, Y/Z)$. As can be seen, the mapping from affine to projective coordinates does not require any operations but the mapping from projective to affine coordinates requires two multiplications and one inversion. Using projective coordinates is very advantageous because point additions and point doublings can be performed without inversions and the total number of inversions in elliptic curve point multiplication is therefore one.

A very efficient algorithm for computing (1) on elliptic curves over $GF(2^m)$ was presented by Julio López and Ricardo Dahab in [9]. They showed that, when Alg. 1 is used, it suffices to consider only the x-coordinate and the y-coordinate can

be recovered in the end [9]. This leads to a very efficient algorithm with projective coordinates. Point addition $(X_3, Z_3) = (X_1, Z_1) + (X_2, Z_2)$ can be computed as follows: [9]

$$Z_3 = (X_1 Z_2 + X_2 Z_1)^2, \quad X_3 = x Z_3 + X_1 Z_2 X_2 Z_1 \tag{2}$$

where x is the x-coordinate of the base point P in Alg. 1. The cost of point addition is four multiplications, two additions and one squaring. Point doubling $(X_3, Z_3) = 2(X_1, Z_1)$ is even simpler [9]

$$X_3 = X_1^4 + a_6 Z_1^4, \quad Z_3 = X_1^2 Z_1^2 \tag{3}$$

where a_6 is a fixed curve parameter. Thus, point doubling costs two multiplications, four squarings and one addition. The y-coordinate is recovered in the end by computing $x_1 = X_1/Z_1$ and $x_2 = X_2/Z_2$ and then by using the formula [9]:

$$y_1 = \frac{(x_1 + x)\left((x_1 + x)(x_2 + x) + x^2 + y\right)}{x} + y \tag{4}$$

where (x, y) is the base point P. This can be computed with one inversion, ten multiplications, six additions and one squaring.

1.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is a standard of ANSI, IEEE, and NIST, among others. The following description is based on Johnson and others' presentation in [5].

The algorithm operates so that first the user, who is commonly called Alice or A for short, generates two keys, private and public, by performing a key pair generation procedure. Then, she publishes her public key. Alice signs a message by performing a signature generation procedure after which she sends both the message and the attached signature to the receiver who is called Bob, or B for short. Bob can verify the signature on the message by first getting Alice's public key and then by performing the signature verification procedure.

Key pair generation, signature generation and signature verification are consider in the following sections.

1.2.1 Key Pair Generation

Private and public key for an identity A is generated as follows:

$$\begin{aligned} d &\in_R [1, n-1] \\ Q &= dG \end{aligned} \tag{5}$$

where $d \in_R [1, n-1]$ means that d is an integer selected at random from the interval $[1, n-1]$. The integer d is A's private key and Q is A's public key. The computation of (5) requires generation of one random integer and computation of one elliptic curve point multiplication.

1.2.2 Signature Generation

In order to generate a signature for a message $\mathcal{M}$ the identity A computes

$$\begin{aligned} k &\in_R [1, n-1] \\ r &= [kG]_x \pmod{n} \\ e &= \text{SHA-1}(\mathcal{M}) \\ s &= k^{-1}(e + dr) \pmod{n} \ . \end{aligned} \tag{6}$$

A's signature on $\mathcal{M}$ is (r, s). The notation $[kG]_x$ denotes the x-coordinate of the result point of kG. Notice that A uses his/her private key d in the signature generation. Thus, other identities cannot produce the same signature without knowing d. Signing a message requires generation of one random integer, computation of one elliptic curve point multiplication and one hashing. In addition, modular inversion, addition and multiplication are required.

1.2.3 Signature Verification

Identity B verifies A's signature (r, s) on the message $\mathcal{M}$ by computing

$$\begin{aligned} e &= \text{SHA-1}(\mathcal{M}) \\ w &= s^{-1} \pmod{n} \\ u_1 &= ew \pmod{n} \\ u_2 &= rw \pmod{n} \\ v &= [u_1 G + u_2 Q]_x \pmod{n} \end{aligned} \tag{7}$$

where Q is A's public key and thus known by B. If $v = r$, B accepts the signature, otherwise (s)he rejects it. Verification requires one hashing and two elliptic curve point multiplications which are combined with a single elliptic curve point addition. Modular inversion and two multiplications are needed, as well.

2 Hardware Architecture

Based on the description of ECDSA given in Sec. 1.2, it is clear that the implementation of ECDSA must be capable of performing the following operations:

- Elliptic curve point multiplication
- SHA-1 hash function
- Modular addition, multiplication and inversion
- (Pseudo-)random number generation

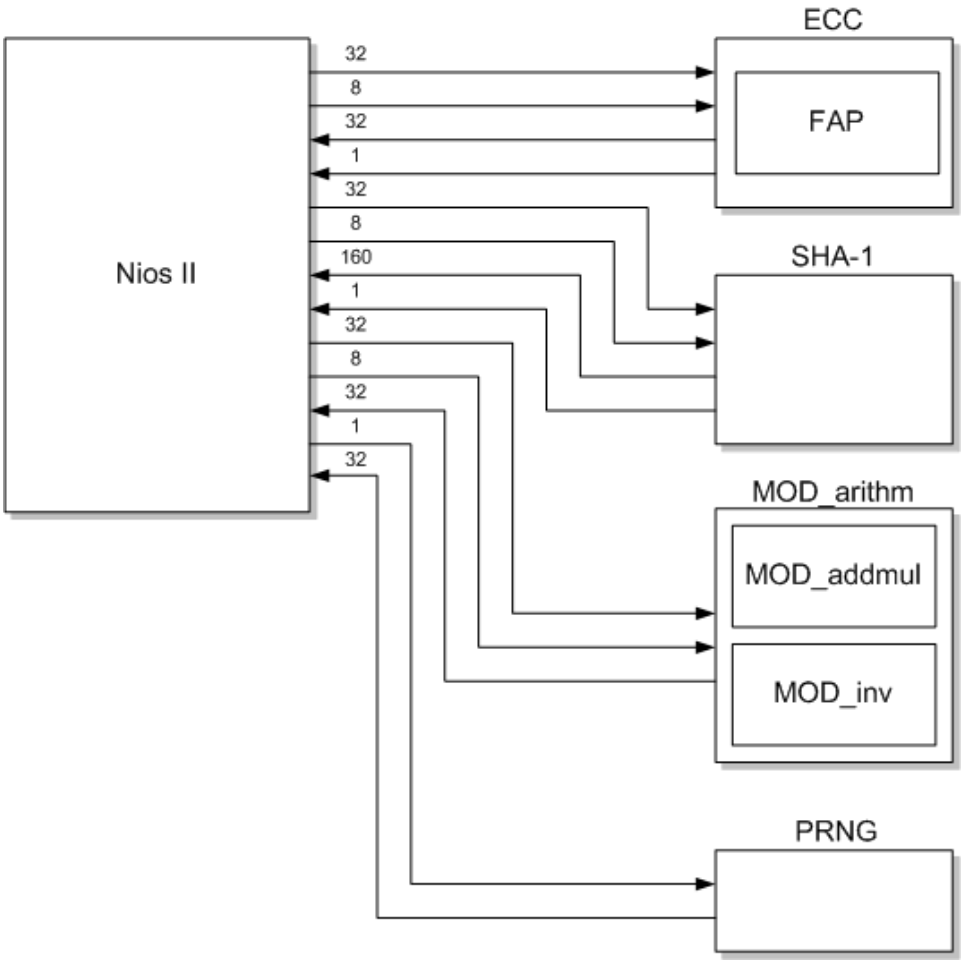

Figure 1: Block diagram of the system

The implementation includes custom-build blocks for each of the above operations in order to ensure fast performance. The most time and resource consuming operation is elliptic curve point multiplication and thus most of the effort was devoted in optimizing it.

Fig. 1 shows the block diagram of the ECDSA system. The Nios II processor is used for user interface and control. Four peripheral components are attached to Nios II and the actual ECDSA computations are performed with them. The peripheral components are elliptic curve module (ECC in Fig. 1), hash module (SHA-1), modular arithmetic module (MOD_arithm), and pseudo-random number generator (PRNG) and they are considered in the following sections. The user interface was realized on Nios II by using Nios II IDE 6.0, and it is considered in Sec. 3.

In order to enhance performance a phase-locked loop (PLL) is used for generating different clocks for different parts of the design. Nios II runs at 50 MHz,

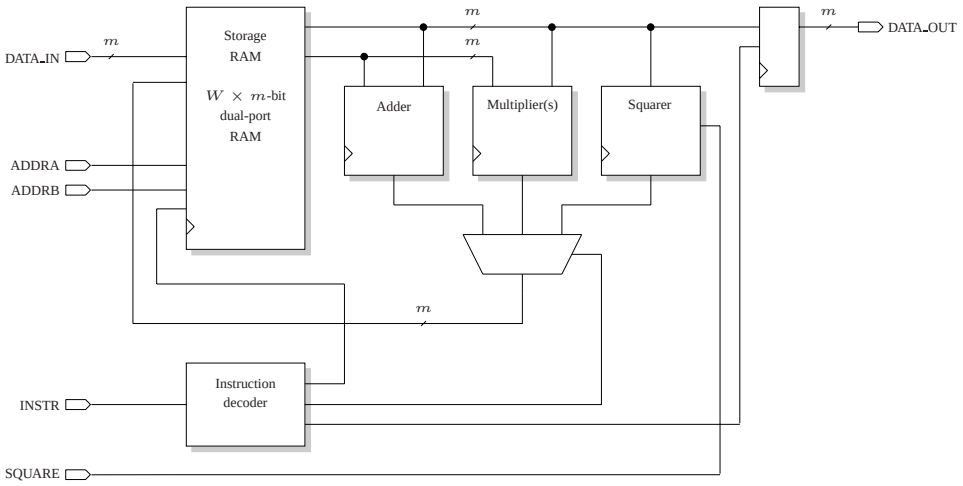

Figure 2: Block diagram of the FAP

ECC, PRNG and SHA-1 blocks at 75 MHz and modular arithmetic blocks at 20 MHz.

2.1 Elliptic Curve Module

The elliptic curve module consists of a field arithmetic processor (FAP) and logic controlling it. That is, the FAP performs operations in $GF(2^{163})$ and the control logic implements elliptic curve operations by using the FAP for field operations. The architecture is based on an elliptic curve processor which has been used in [2, 7, 8] which are scientific publications (co-)authored by the author. The author is alone responsible for the development of the architecture and all VHDL coding.

2.1.1 Field Arithmetic Processor

The FAP consists of adder, squarer, multiplier, storage RAM and instruction decoder. Block diagram of the FAP is presented in Fig. 2.

Adder and Squarer The adder computes a bitwise XOR of two m-bit operands, and it has a latency of one clock cycle. The squarer supports computation of multiple successive squarings, i.e. x^{2^d} where x is an element of $GF(2^{163})$ and d is an integer in the interval $[0, d_{\max}]$ with $d_{\max} = 2^5 - 1$. In normal basis squaring is a rotation of the bit vector as mentioned in Sec. 1.1, and the squarer is a shifter which computes x^{2^d} in one clock cycle.

Multiplier Field multiplication is critical for the overall performance. Multiplication in normal basis is performed with a multiplier which is a digit-serial implementation of the Massey-Omura multiplier [14]. In a bit-serial Massey-Omura

multiplier, one bit of the output is calculated in one clock cycle and, hence, m cycles are required in total. One bit z_i of the result $z = x \times y$, where x, y, z are elements of $GF(2^{163})$, is computed from x and y by using an F-function. The F-function is field specific, and the same F is used for all output bits z_i as follows: $z_i = F(x_{\lll i}, y_{\lll i})$, where $\lll i$ denotes cyclical left shift by i bits. Hence, a bit-serial implementation of the Massey-Omura multiplier requires three m-bit shift registers and one F-function block. A bit-parallel implementation, where all bits z_i are computed in parallel in one clock cycle, requires m F-function blocks and an m-bit register for storing the result. [11, 14]

In practice, the bit-serial implementation requiring at least $m + 1$ clock cycles is too slow and the bit-parallel implementation requires too much area. A good tradeoff is a digit-serial multiplier, where p bits are computed in parallel with p F-function blocks. The F-function blocks can be pipelined in order to increase the maximum clock frequency. As one clock cycle is required in loading the operands into the shift registers and each pipeline stage increases latency by one clock cycle, the latency becomes

$$\left\lceil \frac{m}{p} \right\rceil + c + 1 \tag{8}$$

where $\lceil \cdot \rceil$ denotes rounding up to the nearest larger integer and c is the number of pipeline stages inside the F-function blocks, i.e. $c \geq 0$. In this design, the parameters were selected to be $c = 1$ and $p = 12$.

Others The storage RAM is used for storing elements of $GF(2^{163})$ and it is implemented as a dual-port RAM by using embedded memory blocks in the FPGA, i.e. M4K blocks. The storage RAM is capable of storing W elements. When the architecture is implemented in a Cyclone II FPGA, a logical choice is $W = 256$ because, while in true dual-port mode, the widest mode that an M4K block can be configured to is 256×18-bits. Thus, the storage RAM requires $\lceil 163/18 \rceil = 10$ M4Ks resulting in a storage capacity of 256×163-bits. This much storage space is rarely needed, but it can be used for example for storing pre-computed points, and selecting a smaller depth than 256 would not reduce the number of required M4Ks. Both writing and reading to and from the storage RAM require one clock cycle. However, the dual-port RAM can be configured into the read-during-write mode [1] which saves certain clock cycles; see Sec. 2.1.2.

The instruction decoder decodes instructions to signals controlling the FAP blocks.

2.1.2 Control Logic

The logic controlling the FAP consists of finite state machine (FSM) and ROM containing instruction sequences.

The instruction sequences are carefully hand-optimized, and certain tricks are used in order to minimize latencies of point operations. As mentioned in Sec 2.1.1,

the read-during-write mode can be used for reducing latencies. In order to maximize the advantages in this case, operations are ordered so that the result of the previous operation is used as the operand of the next operation whenever possible. This saves one clock cycle because the operands of the next operation can be read simultaneously while the result of the previous operation is being written.

Inversions in $GF(2^{163})$ are computed with successive multiplications and squarings as suggested by Itoh and Tsujii in [4]. An Itoh-Tsujii inversion has the constant cost of 9 multiplications and 162 squarings when $m = 163$ [4]. Although the number of squarings is high, the successive squaring feature of the squarer (see Sec. 2.1.1) ensures that the computational cost of the squarings remains reasonable.

The elliptic curve module computes point addition and point doubling (one step in Alg. 1) in 125 clock cycles. Interfacing and mapping back to affine coordinates requires 404 clock cycles. In total 162 point additions and point doublings are needed because $\ell = 163$ in Alg. 1. Thus, one elliptic curve point multiplication requires 20,654 clock cycles. Signature verification requires computation of two elliptic curve point multiplications whose results are added with a point addition. This point addition is performed in affine coordinates and it requires 247 clock cycles and the elliptic curve operations computed in verification thus require 41,555 clock cycles.

2.2 Hash Module

The hash module implements SHA-1 hash algorithm according to the standard [12]. The architecture is described in detail in [6] but a short review is given here.

First of all, SHA-1 handles messages in blocks of 512 bits each of which requires computation of 80 steps. One step handles five 32-bit variables by computing four 32-bit modular additions ($a + b \bmod 2^{32}$) and certain 32-bit logical functions which depend on the step index. When all blocks have been processed, the hash of the message is in the five variables and thus SHA-1 outputs a 160-bit hash. [12]

The hash module implements SHA-1 in a straightforward manner and the design utilizes only logic resources. The VHDL describing the design is portable and device independent[1].

The implementation consists of four main components; namely, step function, message schedule, constants block and control logic. The step function block determines the maximum clock cycle of the implementation and it was carefully optimized. The four 32-bit additions form the critical path and all other operations (logic operations and rotations) are computed in parallel with these additions. The message schedule stores 512-bits message bits and derives a 32-bit word for each step from these 512 bits by using three 32-bit bitwise XORs and a rotation. The constants block includes step constants. The word from the message schedule and

[1] Actually, the code was originally written for Xilinx Virtex-II FPGA but it synthesized for Cyclone II without any modifications.

the constants are then used in the step function. The control logic is used for controlling the computation and it consists of a counter, multiplexors and coders. [6]

2.3 Modular Arithmetic Modules

Modular arithmetic modules implement the following operations:

- $c = a + b \pmod{n}$
- $c = a \times b \pmod{n}$
- Computation of a^{-1} such that $a^{-1} \times a = 1 \pmod{n}$

where a, b, c are integers in the interval $[0, n-1]$ and n is a fixed prime number which is hardwired into the design. The length of all integers is 163 bits.

Addition and multiplication are implemented in the same block. Modular addition is trivial. Addition is first carried out as a traditional 163-bit integer addition. If the result is larger or equal to n, then n is subtracted from the result. An addition is computed in two clock cycles.

Modular multiplication is more involved. In this design, it is carried out as shown in Alg. 2 of the Appendix. Computation of $2^i b$ is a shift to the left. The register holding a is shifted to the right so that it suffices to observe only the lsb of the register. One step in the for-loop in Alg. 2 requires three clock cycles and thus multiplication requires 489 clock cycles.

Inversion is the most complex operation of the three. It is performed with a binary algorithm as presented in [3], for example. The algorithm is presented in Alg. 3 of the Appendix. The binary algorithm was chosen because it does not require any integer divisions and is therefore efficient to implement. The latency of inversion is not constant but on average it is about the same as the latency of multiplication. Inversion, however, requires significantly more resources than multiplication.

2.4 Pseudo-Random Number Generator

A pseudo-random number generator (PRNG) is used in generating random integers in key pair generation and signing. The PRNG was implemented as a Linear Feedback Shift Register (LFSR). The length of the shift register was chosen to be 128 bits, and an irreducible polynomial [13]

$$p(x) = x^{128} + x^7 + x^2 + x + 1 \tag{9}$$

was used as a feedback function. Thus, the LFSR has a period of $2^{128} - 1$ bits [13]. The output bits from the LFSR are stored in a 32-bit shift register whose value is shown as the output of the PRNG every 32nd clock cycle. At the beginning, the PRNG is set to an initial state (all ones) with the reset signal.

```
Elliptic Curve Digital Signature Algorithm (ECDSA)
USER INTERFACE; (c) Kimmo Järvinen, 2007

MAIN MENU:
(g) Generate a new identity.
(s) Sign a message.
(v) Verify a signature.
(p) Print timings.
Enter your selection:
```

Figure 3: Main menu of the user interface

3 User Interface in Nios II IDE

A user interface was created in order to be able to demonstrate the operation of the ECDSA blocks. It should be noticed that the user interface was designed for demonstration purpose only, and it was not optimized for performance. In order to use the blocks in an application requiring fast performance, the user interface (and probably the Nios II processor altogether) should be replaced with custom-written logic.

The user interface was written in C language and it is used with the Nios II IDE. It supports four operations:

- Generation of new identities
- Signing of messages
- Verification of signatures
- Performance evaluation

The above list also forms the main menu of the user interface which is shown in Fig. 3. Three first ones are ECDSA operations and they implement equations given in Sec. 1.2. The last operation is used for measuring the performance of the implementation. A performance counter was attached to the Nios II processor and it can be used for measuring different parts of the code.

The user interface uses host file system supported by the IDE and it stores and handles identities, messages and signatures which are located on the harddisk of the host computer. This slows down the performance of the user interface but this approach was chosen because of the ease of implementation and use.

The C code is structured so that `ecdsa_interface.c` describes the user interface and ECDSA functions are given in `ecdsa.h/c`. The ECDSA functions directly control the peripheral components attached to Nios II. The functions include both top level functions for performing high-level tasks such as signature generation and verification and low-level handles for controlling each component

Table 1: Area consumption

Component	LEs	Regs.	M4Ks
Nios II	2,879	1,715	14
ECC	6,441	3,696	22
SHA-1	1,855	1,228	0
MOD_addmul	1,547	542	0
MOD_inv	2,871	1,026	0
PRNG	199	197	0
Total	15,879	8,472	36

individually. There are also handles for starting and stoping operations so that computations can be easily parallelized. For example, one can first begin computation of elliptic curve point multiplication which is the most time consuming operation, then begin hash function, compute some modular arithmetic operations after which different handles can be used for collecting the results of hashing and point multiplication. Considerable performance increases can be achieved with this approach compared to computing all operations sequentially.

The ECDSA functions are fast enough to be used also in real applications. However, custom-written control logic is probably needed for applications requiring very fast performance because Nios II will become the bottleneck as it already slows down the performance considerably as will be shown in Sec. 4.

4 Results and Discussion

The architecture described in Sec. 2 was written in VHDL and synthesized for Cyclone II EP2C20F484C7 with Quartus II 6.0 SP1. ModelSim SE 6.1b was used for simulating the code. Table 1 presents the area consumption of the design components. The design occupies 85 % of the logic elements (LEs) available on the device. Memory block (M4K) usage is 69 %.

Timing evaluations can be computed based on theoretical values and by measuring them with the performance counter. The theoretical values represent the time that the hardware module computes an operation whereas values given by the performance counter always include some overhead caused by Nios II. Thus, both of these values are provided in Table 2. Measured timings in Table 2 are averages from five runs. End-to-end time includes printings to the console and communication with the host computer using host file system. Computation time is the time consumed for computation of the ECDSA operations. ECC only time denotes the time that is taken by elliptic curve operations. Theoretical computation time assumes that parallel computation is used for all operations whenever possible. The time required in interfacing is neglected in theoretical times but included in measured times, and theoretical times equal with the actual computation times without

Table 2: Timings in milliseconds

Operation	Theoretical	Measured
Key pair generation		
End-to-end	n/a	1143.32
Computation	0.28	**0.60**
ECC only	0.28	0.50
Signature generation		
End-to-end	n/a	2073.84
Computation	0.35	**0.94**
ECC only	0.28	0.54
Signature verification		
End-to-end	n/a	2296.73
Computation	0.67	**1.61**
ECC only	0.55	1.00

time spent in interfacing. Notice that the bolded values in Table 2 should be used for comparisons to other designs because the user interface was designed only for demonstration purposes.

Key pair generation is expectedly the fastest operation because it requires only one point multiplication and generation of a random integer. Verification which requires computation of two point multiplications is, again, expectedly the slowest operation. Elliptic curve point multiplication dominates in computation of all three operations.

5 List of Possible Improvements

- Modular arithmetic components are currently straightforward implementations of simple algorithms and considerable increases in performance and reductions in area would probably apply if more efficient algorithms were implemented.
- The two elliptic curve point multiplications in verification could be accelerated by using multiple point multiplication techniques. In these techniques, both point multiplications are computed simultaneously resulting in considerable increase in speed.
- Koblitz curves could be used instead of general elliptic curves. This would speed up elliptic curve operations by approximately 50 % as shown in [8].

References

[1] Altera Corporation. Cyclone II device handbook, February 2007.

[2] V.S. Dimitrov, K.U. Järvinen, M.J. Jacobson, W.F. Chan, and Z. Huang. FPGA implementation of point multiplication on Koblitz curves using Kleinian integers. In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems, CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 445–459, Yokohama, Japan, October 10–13, 2006.

[3] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[4] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78(3):171–177, September 1988.

[5] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, August 2001.

[6] K. Järvinen. Design and implementation of a SHA-1 hash module on FPGAs. Technical report, Helsinki University of Technology, Signal Processing Laboratory, November 2004. `http://wooster.hut.fi/~kjarvine/documents/sha.pdf`.

[7] K. Järvinen, J. Forsten, and J. Skyttä. FPGA design of self-certified signature verification on Koblitz curves. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems, CHES 2007*, Vienna, Austria, September 10-13, 2007. To appear.

[8] K. Järvinen and J. Skyttä. On parallelization of high-speed processors for elliptic curve cryptography. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2007. Submitted.

[9] J. López and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems, CHES 1999*, volume 1717 of *Lecture Notes in Computer Science*, pages 316–317, Worcester, Massachusetts, USA, August 12–13, 1999.

[10] P. L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, January 1987.

[11] National Institute of Standards and Technology (NIST). Digital signature standard (DSS). *Federal Information Processing Standard, FIPS PUB 186-2*, January 27, 2000.

[12] National Institute of Standards and Technology (NIST). Secure hash standard (SHS). *Federal Information Processing Standard, FIPS PUB 180-2*, August 1, 2002.

[13] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 2nd edition, 1996.

[14] C. C. Wang, T. K. Troung, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. S. Reed. VLSI architectures for computing multiplications and inverses in $GF(2^m)$. *IEEE Transactions on Computers*, 34(8):709–717, August 1985.

[15] T. Wollinger, J. Guajardo, and C. Paar. Security on FPGAs: State-of-the-art implementations and attacks. *ACM Transactions on Embedded Computing Systems*, 3(3):534–574, August 2004.

Appendix: Algorithms

Algorithm 1 Point multiplication using Montgomery's ladder

Require: Point P, integer $k = \sum_{i=0}^{\ell-1} k_i 2^i$ where $k_i \in \{0, 1\}$ and $k_{\ell-1} = 1$
Ensure: Point $Q = kP$
  $P_1 \leftarrow P$ and $P_2 \leftarrow 2P$
  for $\ell - 2$ **downto** 0 **do**
    if $k_i = 0$ **then**
      $P_1 \leftarrow 2P_1$ and $P_2 \leftarrow P_1 + P_2$
    else
      $P_1 \leftarrow P_1 + P_2$ and $P_2 \leftarrow 2P_2$
    end if
  end for
  $Q \leftarrow P_1$

Algorithm 2 Modular addition, $c = a \times b \bmod n$

Require: Two ℓ-bit integers a and b in the interval $[0, n-1]$ and a prime n
Ensure: $c = a \times b \pmod{n}$
  $c \leftarrow 0$
  for $i = 0$ **to** $\ell - 1$ **do**
    if a is odd **then**
      $c \leftarrow c + b$
    end if
    $a \leftarrow \lfloor a/2 \rfloor$ $\{a \ggg 1\}$
    $b \leftarrow 2b$ $\{b \lll 1\}$
    if $b \geq n$ **then**
      $b \leftarrow b - n$
    end if
    if $c \geq n$ **then**
      $c \leftarrow c - n$
    end if
  end for

Algorithm 3 Modular inversion, $c = a^{-1} \bmod n$

Require: An integer a in the interval $[1, n-1]$ and a prime n
Ensure: $c = a^{-1} \pmod{n}$

```
u ← a, v ← n
x1 ← 1, x2 ← 0
while u ≠ 1 and v ≠ 1 do
  while u is even do
    u ← u/2
    if x1 is even then
      x1 ← x1/2
    else
      x1 ← (x1 + n)/2
    end if
  end while
  while v is even do
    v ← v/2
    if x2 is even then
      x2 ← x2/2
    else
      x2 ← (x2 + n)/2
    end if
  end while
  if u ≥ v then
    u ← u − v, x1 ← x1 − x2
  else
    v ← v − u, x2 ← x2 − x1
  end if
end while
if u = 1 then
  if x1 ≥ 0 then
    c ← x1
  else
    c ← x1 + n
  end if
else
  if x2 ≥ 0 then
    c ← x2
  else
    c ← x2 + n
  end if
end if
```