Compact ECDSA engine for IoT applications

Article *in* Electronics Letters · June 2016

DOI: 10.1049/el.2016.0760

CITATIONS
11

READS
260

1 author:

Tolga Yalcin
Northern Arizona University
30 PUBLICATIONS **863** CITATIONS

SEE PROFILE

Compact ECDSA Engine for IoT Applications

Tolga Yalçın

Security problems introduced with rapid increase in deployment of IoT devices can be overcome only with lightweight cryptographic schemes and modules. This work presents a compact prime field ($GF(p)$) elliptic curve digital signature algorithm (ECDSA) engine suitable for use in such applications. Generic architecture of the engine makes it suitable for other elliptic curve based schemes (EC Diffie-Hellman key exchange, EC integrated encryption, EC factoring, etc.) with slight modifications. The presented engine is composed of a simple microcoded controller and application specific processing units. It can work with elliptic curves of up to 256-bits, while 160-bit ECDSA signature generation takes 490K cycles. In this work, the engine is implemented as an IP in a 180 nm process. However, its architecture allows it to be implemented on any ASIC or FPGA platform with dual-port memory support. In view of its gate count of 11366 GEs, the presented work is the most compact ECDSA engine with capability for a wide-range of curves and different applications.

Introduction: Since their introduction in 1985, elliptic curves have been the de-facto alternative in public-key cryptography. Their smaller key sizes have made them unbeatable in real-world implementations and their use have soared in the last decade. As an example, 160-bit prime elliptic curves provide security equivalent to that of 1024-bit RSA, which in fact is equivalent to 80-bit symmetric key security [1]. Several discrete logarithm based public-key schemes have been adapted to elliptic curves. Among these, elliptic curve digital signal algorithm (ECDSA) [2] is of particular interest due to wide spread use of digital signatures in modern communication systems. ECDSA offers fast processing times and low implementation costs.

As with the DSA, ECDSA incorporates two operations, signature generation and signature verification. The pseudo-codes for both operations are given in Algorithms 4.29 and 4.30 in [3]. The multiplication and addition operations used in the computation of kP and X all take place on the elliptic curves, and are referred to as point multiplication and point addition, respectively. Point multiplication is performed via point addition and point doubling (not shown in the algorithms), which form the basic operations of elliptic curve cryptography. They are both realized by modular arithmetic operations (addition, subtraction, multiplication and inversion).

Modular addition and subtraction are straightforward, whereas modular multiplication can be performed either by schoolbook multiplication followed by modular reduction, or by Montgomery's modular multiplication algorithm [4]. Modular inversion can be realized by Fermat's Little Theorem which uses modular multiplication, t'hereby eliminating the need for any special inversion block. The basic EC point operations can be performed on various coordinate representations (i.e. Cartesian/affine, Jacobian, projective, etc.) [3]. However, regardless of the coordinate system, each operation heavily relies on modular arithmetic. Therefore, it is essential to realize modular arithmetic operations efficiently on any target platform.

On hardware, elliptic curve cryptography can be implemented with either custom hardware blocks or a custom processor. Usually, a processor based a'further simplify the processor design and reduce it to a single-instruction set computer [7]. In this work, a hybrid of custom hardware blocks (processing units) and a simple microcontroller has been used in order to come up with a simple, low-cost and reconfigurable architecture with decent speed performance. The microcontroller has very limited capability. Its only duty is to assign jobs to the processing units and control data flow. Details of this architecture will be given in the next section, followed by results and conclusion.

ECDSA engine architecture: ECDSA engine incorporates an application-specific elliptic curve cryptography (ECC) microcontroller, various processing units and a dual-port RAM used as the data memory, as shown in Fig. 1. The microcontroller, processing units and dual-port RAM are referred to as MICRO, PUNITS and DPMEM, respectively, in the rest of this article. Engine operation can be summarized as follows:

Input data is first loaded into the data memory through its port-A by the host controller (main controller of the system which the ECDSA engine is a part of). The host controller then sets the input options (operation mode,

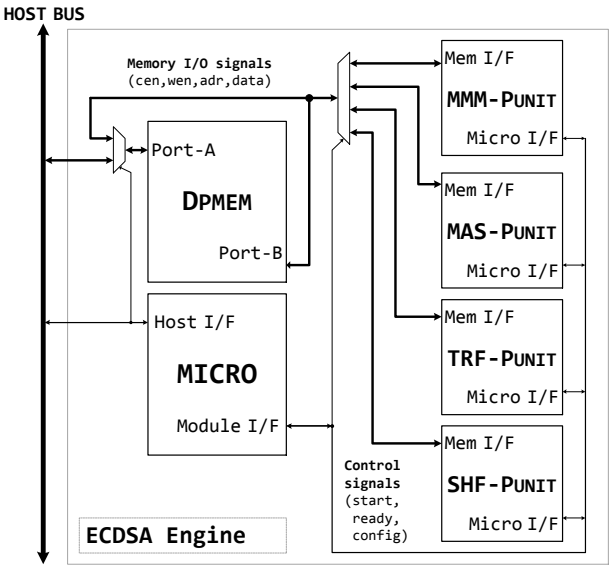

Fig. 1. ECDSA engine architecture

curve type) via the host-controller interface of the engine and sends the "request" signal, which instructs the engine to start its operation.

MICRO (see Fig. 2) sets its program counter (PC) to the address of the subroutine of requested operation (signature generation, verification, etc.) and starts running instructions from its program memory. It has a very simple architecture with only two instructions. Each instruction is run in three cycles – fetch, decode and execute – in a pipelined manner. With each new PC value, a new instruction is fetched from program memory in the fetch cycle. It then is decoded by the instruction decoder and appropriate control signals are generated in the decode cycle. Finally, the decoded instruction is executed in the execute cycle.

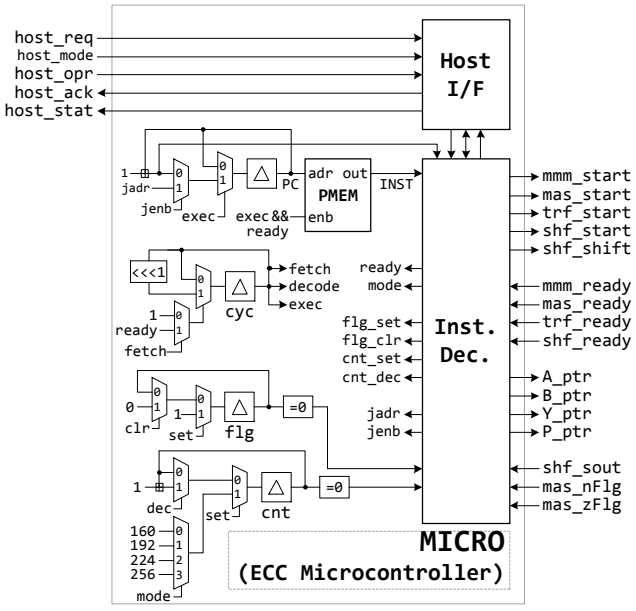

Fig. 2. ECC microcontroller

As stated before, there are only two types of instructions: PUNIT-run and conditional-jump. If the fetched instruction is a PUNIT-run instruction, MICRO sends operand pointers and a "start" signal to the target PUNIT, which then starts its designed operation and takes over control of both ports of DPMEM. At this time, MICRO halts in the execute cycle and waits until the target PUNIT completes its operation.

Datapath width for the PUNITs and DPMEM is 32 bits. Unlike conventional microcontrollers, MICRO has no direct access to the data memory. Instead, it instructs the PUNITs to work on data stored in the memory addressed by the pointers. Variables (EC coordinates, coefficients, constants, etc.) are stored in DPMEM in a little-endian word order (least significant word first). Array sizes depend on the selected operation mode (curve size). For example, for ECC-160 mode (160-bit prime curves), array size is 160 bits (5 words). Start address of an array is determined by multiplying its pointer with number of words per array (5, 6, 7 and 8 for 160, 192, 224 and 256-bit curves, respectively). PUNIT in charge reads and writes DPMEM through both its ports. Upon completion of its operation, it releases memory ports and sends a "ready" signal acknowledging MICRO, which then proceeds with fetching of the next instruction.

In case of a conditional-jump instruction, MICRO checks the condition and makes jump decision during decode cycle. It then sets a flag and executes the jump in the execute cycle. The format of this instruction is "**if** *condition*, **set** *flag*, **goto** *address*". In order to provide maximum flexibility, *unconditional*, *no-flag* and *next-line* are also implemented as condition, flag and jump address, respectively. For example, an unconditional jump without any flags can be executed by "**if** *unconditional*, **set** *no-flag*, **goto** *address*". Similarly, a flag can be set and proceeded to the next line by "**if** *unconditional*, **set** *flag*, **goto** *next-line*". At every successful jump, the *next-line* address is pushed into the "stack", which is a part of the instruction decoder and has a depth of 2.

Each instruction is 16 bits wide and the program memory (implemented using combination logic) has a depth of 215 lines to implement the complete ECDSA suite. PUNIT-run instruction uses 1 bit for opcode, 12 bits for operands and result pointers (4 bits each) and 3 bits for PUNIT selection. Conditional-jump instruction uses 1 bit for opcode, 3 bits for jump condition select, 4 bits for status bit select and 8 bits for jump address. In the present implementation, Jacobian coordinates are used together with affine coordinates in order to achieve the lowest multiplication count. Size and implementation information for each part of the code is summarized in Table 1.

Table 1: ECDSA code information

Subroutine	No. of lines	Implementation
main	3	Initializations
Signature generation (siggen)	52	Affine coordinates
Signature verification (sigvrf)	94	Affine coordinates
Modular inversion (modinv)	12	Affine coordinates
Point multiplication (ptmlt)	14	Affine coordinates
Point doubling (ptdbl)	21	Jacobian coordinates
Point addition (ptadd)	19	Hybrid (Jacobian+affine)

Once the program execution is completed, MICRO shuts downs all PUNITs, releases port-A of DPMEM, sends "acknowledge" and status signals to the host controller, and starts waiting for the next "request". It should be noted that in the target system, DPMEM is part of host memory.

In the present architecture, there are four PUNITs. These are Montgomery modular multiplication unit (PUNIT-MMM), modular addition/subtraction unit (PUNIT-MAS), memory array transfer unit (PUNIT-TRF) and memory array shift unit (PUNIT-SHF). All PUNITs use minimal number of registers (only a few for operand and/or temporary storage) and rather directly work on data in DPMEM, in order to minimize register area.

PUNIT-MMM performs radix-16 Montgomery modular multiplication using two 4×32-bit multipliers. These multipliers are implemented as simple adder trees. Therefore, no special multiplier modules (such as Xilinx DSP cells) are required, making PUNIT-MMM compatible with any technology. Additionally, PUNIT-MMM incorporates a 8×32-bit shift register, which is used to store recursion values of multiplication result. This way, DPMEM access is minimized and high speed operation is achieved. A single 160-bit Montgomery modular multiplication is completed in 203 cycles. This directly affects the performance of point operations, and consecutively the performance of ECDSA signature generation and verification. In the present implementation, point multiplication on an n-bit curve requires a total of $15n$ modular multiplications and $15.5n$ modular additions/subtractions.

PUNIT-MAS performs modular addition or subtraction on data arrays. It uses only a single 32-bit adder inside.

PUNIT-TRF transfers data arrays from one location to another inside DPMEM. Essentially, it performs $a \leftarrow b$, where both a and b are multi-word data arrays. Although this could be performed within a for-loop word-by-word, high number of these transfers entailed such a unit in order to prevent performance loss.

Table 2: Performance figures and comparison to prior work

Design	Area	Memory	Mults	ECPM cyc
[5]	1158 slices	3 BRAMs	4	950K
This work (160-bit)	937 slices	2 BRAMs	-	490K
This work (multi-mode)	1036 slices	2 BRAMs	-	840K
[6]	12448 GEs	included	-	<140K
This work (160-bit)	11366 GEs	external	-	490K
This work (multi-mode)	12324 GEs	external	-	840K

PUNIT-SHF is used to check bits of a long array within a loop. In other words, it performs **for** $i = n-1$ *downto* 0, **if** $x[i] = 1$ ... check for data arrays. For this, it uses a 32-bit internal shift register onto which it loads the target array from DPMEM word-by-word at every 32 bits and then left shifts the register bit-by-bit. The most significant bit of this register is used as one of the jump conditions of conditional-jump instruction. PUNIT-SHF is used in the execution of point multiplication and modular inversion subroutines. A second version of PUNIT-SHF was also designed in order to convert input arrays to non-adjacent form (NAF) [3] and reduce the point multiplication time. However, this version required a 512-bit shift register to store NAF coefficients, which meant a 78.9% increase of the register count. This was too high compared to 12.2% reduction in the execution time, resulting in dismissal of the NAF implementation.

Results and conclusion: The ECDSA engine is implemented in a 180 nm CMOS technology as a subsystem of a system-on-chip (SoC) for Internet-of-Things (IoT) applications. It is originally designed only for 160-bit curves. However, in order to be able to make a fair comparison with the existing works, a multiple curve (160, 192, 224 and 256-bit) version is also implemented (referred to as "multi-mode"). For the same reason, both designs are implemented on FPGAs as well.

The 160-bit and multi-mode versions occupy 11366 and 12324 gate equivalents (GEs), respectively. At the 10 MHz target frequency of the application, 160-bit signature generation is completed in 49 ms. Detailed performance figures and comparison to prior work is given in Table 2.

The presented design beats the best FPGA area in literature by 10.5%. On the ASIC side, it is 9% smaller (without the DPRAM, which is part of the host system in our application). However, it should be noted that the presented ECDSA engine offers much more flexibility than previous work. It can be modified to perform any other elliptic curve cryptography scheme (such as EC Diffie-Hellman, EC integrated encryption, EC factoring) by only program code modification. Its architecture allows it to be modified not only for larger prime curves, but also for different types of curves (Edwards, Hessian, etc. – both prime and binary) with minor changes to PUNITs. Furthermore, additional PUNITs (a hashing unit or a random number generator) can be plugged into MICRO in order to provide additional functionality.

Acknowledgment: This work has been partially supported by ASELSAN A.Ş.

Tolga Yalçın (*Computer Engineering Department, Food and Agriculture University, Konya, Turkey*)

E-mail: tolga.yalcin@gidatarim.edu.tr

References

1 Standards for Efficient Cryptography: 'SEC 1: Elliptic Curve Cryptography', 2009, p. 73
2 National Institute of Standards and Technology: 'FIPS PUB 186-4: Digital Signature Standard', 2013, p. 6
3 Hankerson D., Menezes A. and Vanstone S.: *Guide to Elliptic Curve Cryptography*, 2004, p. 184
4 Montgomery P. L.: 'Modular multiplication without trial division', *Math. of Comp.*, 1985, **44**, pp. 519-521
5 Varchola M., Güneysu T. and Mischke O.: 'MicroECC: A Lightweight Reconfigurable Elliptic Curve Crypto-processor', *ReConFig*, 2011, pp. 204-210
6 Pessl P. and Hutter M.: 'Curved Tags – A Low-Resource ECDSA Implementation Tailored for RFID', *LNCS*, 2014, **8651**, pp. 156-172
7 Roy D. B., Das P. and Mukhopadhyay D.: 'ECC on Your Fingertips', *Cryptology ePrint Archive*, 2015