A Public-key Cryptographic Processor for RSA and ECC

Hans Eberle, Nils Gura, Sheueling Chang Shantz, Vipul Gupta, Leonard Rarick
Sun Microsystems Laboratories
{hans.eberle, nils.gura, sheueling.chang, vipul.gupta, leonard.rarick}@sun.com

Shreyas Sundaram
University of Waterloo
ssundara@ieee.org

Abstract

We describe a general-purpose processor architecture for accelerating public-key computations on server systems that demand high performance and flexibility to accommodate large numbers of secure connections with heterogeneous clients that are likely to be limited in the set of cryptographic algorithms supported. Flexibility is achieved in that the processor supports multiple public-key cryptosystems, namely RSA, DSA, DH, and ECC, arbitrary key sizes and, in the case of ECC, arbitrary curves over fields $GF(p)$ and $GF(2^m)$.

At the core of the processor is a novel dual-field multiplier based on a modified carry-save adder (CSA) tree that supports both $GF(p)$ and $GF(2^m)$. In the case of a 64-bit integer multiplier, the necessary modifications increase its size by a mere 5%. To efficiently schedule the multiplier, we implemented a multiply-accumulate instruction that combines several steps of a multiple-precision multiplication in a single operation: multiplication, carry propagation, and partial product accumulation.

We have developed a hardware prototype of the cryptographic processor in FPGA technology. If implemented in current 1.5 GHz processor technology, the processor executes 5,265 RSA-1024 op/s and 25,756 ECC-163 op/s - the given key sizes offer comparable security strength. Looking at future security levels, performance is 786 op/s for RSA-2048 and 9,576 op/s for ECC-233.

1 Introduction

In this paper we describe an extension to a general-purpose processor for accelerating public-key cryptosystems. Supported are the legacy cryptosystems RSA and DH as well as the newly emerging Elliptic Curve Cryptography (ECC) system. As we will show, minimal extensions suffice to efficiently support these public-key cryptosystems.

Due to its computational efficiency, ECC is emerging as an attractive alternative to traditional public-key cryptosystems such as RSA, DSA, and DH. More specifically, ECC offers equivalent security with smaller key sizes, in less computation time and with less memory. As a result, ECC offers higher throughput on the server side [9] and smaller implementations on the client side. By saving system resources ECC is particularly well suited for small devices such as mobile phones, PDAs and smart cards.

ECC technology is ready for deployment as, in addition to its technical merits, standards have been put in place and reference implementations have been made available. Several

standards have been created to specify the use of ECC. The US government has adopted the Elliptic Curve Digital Signature Algorithm (ECDSA [1], the Elliptic Curve variant of DSA) and recommended a set of curves [26]. Additional curves for commercial use were recommended by the Standards for Efficient Cryptography Group (SECG) [2]. Also, our group has added ECC functionality to OpenSSL [21] and Netscape Security Services (NSS) [20], the two most widely used open source implementations of the SSL protocol [10]. These cryptographic libraries are used by numerous applications including the Apache web server and the Mozilla/Netscape browsers.

In this paper, we focus on hardware acceleration of public-key cryptosystems on server machines. Servers running security protocols such as SSL or IPsec are confronted with an aggregation of secure connections created by a multitude of heterogeneous clients. Handling a high volume of secure connections on the server side not only demands high computational power but also flexibility in responding to client devices that are limited in the set of cryptographic algorithms supported. We, therefore, made it a goal to support a variety of cryptographic algorithms to be able to communicate with as many clients as possible. More specifically, the processor had to support the legacy cryptosystem RSA in addition to the new cryptosystem ECC, it further had to support arbitrary key sizes for any one of these algorithms, and with respect to ECC, it had to provide dual-field arithmetic for fields $GF(p)$ and $GF(2^m)$.

2 Related Work

Most published papers on cryptographic processors describe algorithm-specific implementations. Processors implementing ECC have been reported in [11, 12, 13, 15, 16, 22, 23]. The architectures of these processors are tailored for ECC and do not support any other algorithm such as RSA. Moreover, these implementations are not flexible enough to support the whole set of ECC curves specified in [2, 26]. In particular, many of these processors implement one field type only.

In [11], we described a cryptographic processor that is flexible enough to handle arbitrary elliptic curves over fields $GF(2^m)$. At 6,955 point multiplications per second for NIST curve 163, this is presently the highest performing hardware implementation. The processor has a 256-bit data path and supports curves up to a field degree of 255. It provides hardware optimizations for named curves standardized by NIST [26] and SECG [2] and firmware support for any other generic curves. In subsequent publications [6], we described a technique called partial reduction [5] that improves the performance for generic curves. While our previous papers described a cryptographic coprocessor with a data path tailored for ECC over fields $GF(2^m)$, this paper describes a more flexible design that can be implemented as an extension of a general-purpose processor and that supports ECC over fields $GF(2^m)$ and $GF(p)$ as well as RSA.

There have been only a few attempts at providing support for different public-key cryptosystems including ECC with a common shared hardware architecture. Such efforts have mostly targeted client devices and, in particular, microcontrollers for smart card applications [8]. These processors, however, are optimized for low-power consumption rather than for high performance.

A cryptographic processor for accelerating RSA as well as ECC is described by Satoh and Takano in [25]. While their design is similar in functionality and performance to

ours, their architecture looks rather different in that it specifically targets the implemented algorithms. More specifically, it uses a data path optimized for modular multiple-precision multiplication that uses dedicated memories, registers, and data links. Similarly, control logic is hardwired for the chosen RSA and ECC algorithms. In contrast, our architecture uses a general-purpose data path and microprogrammable control logic.

3 Architecture

Our public-key processor supports both the RSA and ECC cryptosystems. Other algorithms such as DSA or DH could be easily supported through firmware without requiring any hardware modifications.

The RSA algorithm uses *modular exponentiation* which can be implemented through repeated multiplication and squaring. The equivalent core function for the ECC cryptosystem is called *point multiplication*. We use a double-and-add algorithm for point multiplications over fields $GF(p)$ and Montgomery Scalar Multiplication for point multiplications over fields $GF(2^m)$. We use projective coordinates for $GF(2^m)$ [17] and mixed coordinates for $GF(p)$ [3].

In contrast to the cryptographic techniques used for bulk encryption and message digest, public-key cryptosystems are more compute-intense and less data-intense and, thus, well suited for today's processor architectures that favor processing data over moving data, especially if data accesses exhibit locality. Public-key cryptosystems heavily rely on multiplication operations which are typically well supported on general-purpose processors.

3.1 A General-purpose Data Path for Public-key Cryptosystems

To adhere to a general-purpose data path, we had to limit the amount and scope of modifications and additions. Thus, we omitted a number of performance optimizations that we described in [7]. In particular, we did not implement hardware division and we did not provide an optimized squarer for $GF(2^m)$.

ECC requires modular multiple-precision division that is a time-consuming operation on general-purpose processors. The number of division operations needed for ECC point multiplication depends on the chosen coordinate system. If projective coordinates are chosen, only a single division operation is needed[1]. In this case, the division, or more precisely, the inversion can be implemented through a series of multiplications using Fermat's little theorem. For the processor described in [7], replacing the hardware divider by a software implementation added 6% to the execution time of an ECC point multiplication over fields $GF(2^m)$.

General-purpose processors typically have a data path width of 8, 16, 32, or 64 bits and operate on operands that have sizes equal to the data path widths. Thus, the long operands of the RSA and ECC algorithms need to be broken up into smaller words, and the arithmetic operations addition, subtraction, and multiplication need to be implemented as multiple-precision operations. Since we focus on server applications, we decided on a 64-bit architecture.

[1] Based on the number of multiplication and division operations required by the different coordinate representations, it can be determined that affine coordinates are favored if the ratio of the execution times for multiplications t_{mul} and divisions t_{div} is approximately $t_{div}/t_{mul} \leq 6.5$ and that projective coordinates are favored for $t_{div}/t_{mul} > 6.5$.

To reduce the amount of additional logic, we tried to leverage the existing data path as much as possible. For this reason, we use a modified version of an integer multiplier that can also support multiple-precision multiplications for the fields $GF(p)$ and $GF(2^m)$. A dual-field multiplier similar to ours has also been described in [25].

3.2 Arithmetic and Control Processors

Multiple-precision multiplications are the predominant operations used by both the RSA and ECC algorithms[2]. To achieve optimal performance for multiple-precision multiplications, the multiplier has to be kept busy at all times. For this reason, we implemented a dual-issue machine consisting of two processors, an arithmetic processor and a control processor. More specifically, we implemented a VLIW architecture, whereby each instruction word contains an arithmetic instruction and a control instruction. For example, when a multiple-precision multiplication is performed, the arithmetic processor executes multiply-accumulate instructions while the control processor, in parallel, executes loop control instructions. This way, the critical path of the program execution is determined by the arithmetic instructions and the control instructions do not add any execution time.

The multi-issue processor architecture described here can be easily mapped onto a modern superscalar processor architecture. More specifically, the control processor is similar to the branch pipeline and the arithmetic processor is similar to the integer pipeline. The difference is that we statically compile tuples of instructions that are issued in the same cycle, whereas in a superscalar architecture it is decided dynamically which instructions are issued in parallel.

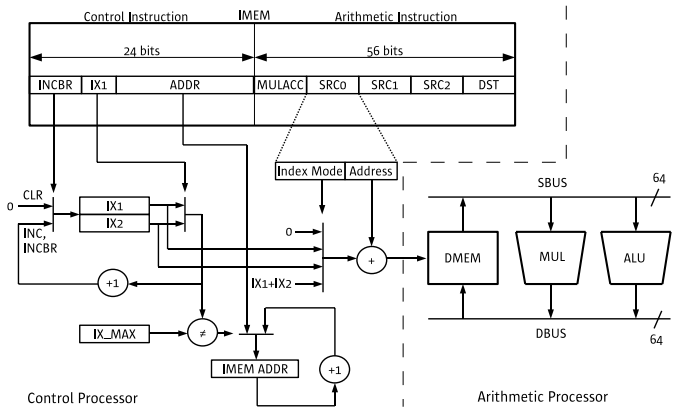

Figure 1. Data path.

Figure 1 shows a block diagram of the processors. The arithmetic processor implements a 64-bit data path consisting of the data memory DMEM and the arithmetic units MUL and ALU. MUL implements multiplication, and ALU implements subtraction and addition - arithmetic operations are implemented for both fields $GF(p)$ and $GF(2^m)$. There is a source bus SBUS and a destination bus DBUS to transfer source and destination operands, respectively, between the DMEM and the arithmetic units MUL and ALU. All operands reside in the DMEM, that is, there is no register file in addition to the memory and instructions access operands directly in memory.

The control processor consists of the instruction memory IMEM and the logic necessary to calculate indices, DMEM addresses and IMEM addresses. A sample instruction tuple {INCBR, MULACC} is shown in Figure 1 to illustrate the operation of the control processor: INCBR stands for *increment and branch*, and MULACC stands for

[2]While the math for ECC is more complicated than for RSA, it is still the case that the multiplications make up the majority of instructions executed as we will show in Section 6.

Table 1. Instruction set.

Arithmetic Instructions		
Instruction Type/Opcode	**Name**	**Description**
ADD SRC0,SRC1,DST	add	DST ← (SRC0+SRC1)[63:0]; ALU.CC ← (SRC0+SRC1)[64]
ADDC SRC0,SRC1,DST	add with carry	DST ← (SRC0+SRC1+ALU.CC)[63:0]; ALU.CC ← (SRC0+SRC1+ALU.CC)[64]
ADDX SRC0,SRC1,DST	XOR add	DST ← (SRC0 ⊕ SRC1)[63:0]; ALU.CC ← 0
SUB SRC0,SRC1,DST	subtract	DST ← (SRC0-SRC1)[63:0]; ALU.CC ← (SRC0-SRC1)[64]
SUBB SRC0,SRC1,DST	subtract with borrow	DST ← (SRC0-SRC1+ALU.CC)[63:0]; ALU.CC ← (SRC0-SRC1+ALU.CC)[64]
MULACC SRC0,SRC1,SRC2,DST	multiply and add	DST ← (SRC0*SRC1+SRC2)[63:0]; MUL.XC ← (SRC0*SRC1+SRC2)[127:64]
MULACCC SRC0,SRC1,SRC2,DST	multiply and add with carry	DST ← (SRC0*SRC1+SRC2+MUL.XC)[63:0]; MUL.XC ← (SRC0*SRC1+SRC2+MUL.XC)[127:64]
MULACCX SRC0,SRC1,SRC2,DST	XOR multiply and add	DST ← ((SRC0 ⊗ SRC1) ⊕ SRC2)[63:0]; MUL.XC ← ((SRC0 ⊗ SRC1) ⊕ SRC2)[127:64]
MULACCXC SRC0,SRC1,SRC2,DST	XOR multiply and add with carry	DST ← ((SRC0 ⊗ SRC1) ⊕ SRC2 ⊕ MUL.XC)[63:0]; MUL.XC ← ((SRC0 ⊗ SRC1) ⊕ SRC2 ⊕ MUL.XC)[127:64]
NOPA	no arithmetic operation	

Control Instructions		
Instruction Type/Opcode	**Name**	**Description**
CLR IX	clear index register	IX ← 0
INC IX	increment index register	IX ← IX+1
INCBR IX,ADDR	increment index register	branch to ADDR if IX ≠ RC.IX_MAX; IX ← IX+1
BCS ADDR	branch if carry set	branch to ADDR if ALU.CC==1
JMP ADDR	jump	jump
END	end	end program execution
NOPC	no control operation	

Symbol Definitions	
Symbol	**Description**
←	assignment
==	equal
≠	not equal
+	addition
−	subtraction using 2's complement
*	multiplication
⊕	addition/subtraction over $GF(2^m)$, equivalent to bitwise XOR
⊗	multiplication over $GF(2^m)$

multiply and accumulate - instructions will be explained in detail in the next section. The control processor translates these instructions into the following operations: increment the index register, compare the values of the index register and the upper bound of the loop index stored in the IX_MAX register to decide whether a branch has to be taken and calculate the addresses of the source and destination operands of the MULACC instruction by adding the value of the index register to the address provided by the instruction.

3.3 Instruction Set

Instructions are fetched in tuples consisting of an arithmetic instruction and a control instruction. The two instructions of a tuple are executed in parallel; there are no data dependencies between them. If instructions of a tuple refer to processor state such as condition code registers or index registers, then the processor's state before the execution of the current tuple is referred to. NOP instructions are provided to fill instruction slots for which no executable instructions are available.

Table 1 lists the arithmetic instructions. The listed instructions implement addition, subtraction, and multiplication. Instructions are provided for operations over fields $GF(p)$ and $GF(2^m)$: ADD, ADDC, SUB, SUBB, MULACC, MULACCC specify operations over fields $GF(p)$ and ADDX, MULACCX, MULACCXC stand for operations over fields $GF(2^m)$[3].

The arithmetic instructions specify up to three source operands and one destination operand. Operands are specified by an instruction field consisting of an index field and

[3]Addition and subtraction over fields $GF(2^m)$ are equivalent to an XOR operation.

an address field. The index field selects one of four index values: $0, IX1, IX2, IX1 + IX2$ where $IX1$ and $IX2$ refer to index registers. The address of the operand is calculated as the sum of the address field and the index value. Index value 0 provides a way to specify absolute addresses. The availability of two index registers allows for implementing loops with a nesting depth of two as needed for multiple-precision multiplication.

The arithmetic units contain carry logic and registers that allow for efficiently implementing multiple-precision arithmetic. More specifically, ALU.CC[0] holds the carry and borrow bit needed for multiple-precision addition and subtraction, respectively. And MUL.XC[63:0] contains the carry bits needed for multiple-precision multiplication - we refer to these bits as *extended carry*.

The control instructions determine the flow of program execution. There are basically two types of control instructions: branch instructions and instructions to manipulate the index registers. The former type includes the conditional branch instructions BCS and INCBR, and the unconditional branch instruction JMP. The latter type consists of instructions CLR and INC. INCBR could also be included in this type as it also manipulates the index registers. Finally, there is instruction END to mark the end of program execution.

We optimized the formats of the arithmetic and control instructions to allow for simple decoding logic rather than for high code density. For example, we decided on fixed instruction lengths and instruction fields with fixed locations. As a result, some instructions contain empty fields that could be avoided with a more flexible format. Further, shorter operand fields could be used if register operands rather than memory operands were used. We decided on memory operands because they simplified access to multiple-precision operands by avoiding management of a register file. Given the FPGA implementation technology, the lack of registers does not reduce performance since accessing a register file is not significantly faster than accessing memory.

The size of the code for ECC point multiplication ranges from 6 to 9.5 kBytes depending on the key size and the field type. For Montgomery modular exponentiation, the size of the code is 1 kByte independent of the key size. The code sizes clearly reflect the regularity of the algorithms: Whereas roughly 100 instruction tuples are needed for Montgomery modular exponentiation, about 1,000 instruction tuples are required for ECC point multiplication over fields $GF(2^m)$[4]. Typical implementations of the RSA algorithm use the Chinese Remainder Theorem (CRT) [18, 24]. With this technique, a modular exponentiation is split into two smaller exponentiation operations using operands for the base and exponents that are both half the size of the original operands. While CRT drastically reduces computation time, it requires a significant amount of instructions in addition to the ones needed for Montgomery modular exponentiation.

There are some obvious techniques to reduce the code size further, in addition to applying a denser instruction format. The code size for multiple-precision arithmetic can be optimized by using hardwired state machines to implement multiple-precision operations rather than using microcode. Alternatively, subroutines could be provided to implement multiple-precision arithmetic operations and other routines.

3.4 A Reduced Instruction Set for Public-key Cryptography

Revisiting the instruction set we notice that a sparser set could actually be implemented. The minimum set of instructions would include five arithmetic instructions and four con-

[4]ECC point multiplications over fields $GF(p)$ take about 10% less instructions.

trol instructions for a total of nine instructions. The arithmetic instructions are: ADDC, ADDX, SUBB, MULACCC, MULACCXC; and the control instructions are: CLR, INCBR, BCS, END. The omitted instructions can be substituted as follows. The arithmetic instructions ADD, SUB, MULACC, MULACCX can be replaced by the corresponding instructions that input the carry bits. This requires that the carry is cleared first, for example, by executing an arithmetic operation that is guaranteed to generate a value 0 for the carry. The arithmetic no op instruction NOPA can be implemented by any arithmetic instruction that does not alter state on which the program depends. The control instructions INC can be replaced by INCBR in that the jump address is set to the address of the next instruction. And finally, the control instructions JMP and NOPC can be replaced by BCS. To implement an absolute jump, the carry tested by BCS has to be set by an arithmetic instruction. And to realize NOPC, BCS with a jump address set to the address of the next instruction had to be used.

Examining the point multiplication code we realize that the performance penalty for using the reduced instruction set is negligible. The arithmetic instructions ADD, SUB, MULACC, and MULACCX that do not input carry bits are mostly used at the beginning of multiple-precision arithmetic operations. These instructions can be easily replaced by the corresponding instructions that do input the carry bits without the need to add instructions that explicitly clear the carry bits. The reason is that the carry bits are left cleared at the end of the previous operations. Implementations relying on this invariant do not incur any overhead when using the reduced instruction set.

Looking at the instruction set needed to support public-key cryptography we note that most instructions are already implemented by general-purpose processors. The only extensions needed are support for MULACC and MULACCX. As we will show in Section 5, existing multiplier designs can be easily modified to provide the necessary functionality.

3.4.1 Implementation

As a proof of concept, we have prototyped the cryptographic processor in a Xilinx Virtex-II XC2V6000 FPGA. The FPGA is interfaced to the host system via a PCI bus. The 66 MHz PCI clock serves as the main clock of the processor. Figure 2 shows the floorplan of the design.

We wrote a simple assembler that allowed us to develop firmware for the processor. The assembler is also used to instrument the code in that it provides counts of the executed instructions. On the host side, the processor is interfaced by a driver for the Solaris^{TM} operating system and a cryptographic library.

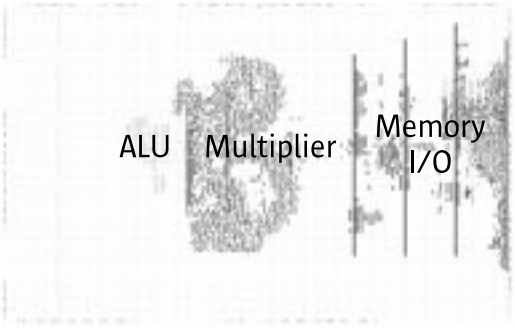

Figure 2. Floorplan.

4 Modular Multiplication

Modular multiplication is the most critical operation underlying both the RSA and ECC cryptosystems. Since both cryptosystems operate on operands that are wider than the 64-bit data path, efficient multiple-precision operations need to be devised. We will first exam-

ine how modular multiplication is implemented as a multiple-precision operation. Next, we will describe a multiplier architecture that efficiently supports multiple-precision modular multiplication.

4.1 Montgomery Modular Multiplication

Modular multiple-precision multiplications underlie both the RSA and the ECC algorithms. An efficient multiplication method is the Montgomery multiplication [19] that replaces divisions with multiplications to compute the reduction. Montgomery multiplication of two integer operands A and B is defined as $C = A * B * r^{-1} \ mod \ M$. For multiple-precision Montgomery multiplication of binary numbers, r equals 2^m with m being the number of bits in M rounded up to the next multiple of the word size. For example, for a 160-bit M on a 64-bit architecture, m would be $m = 3 * 64 = 192$. Similarly, Montgomery multiplication can be applied to polynomials $A(t)$, $B(t)$, and $M(t)$: $C(t) = A(t) * B(t) * r^{-1} \ mod \ M(t)$, where r is typically t^m with m being the degree of M rounded up to the next multiple of the word size.

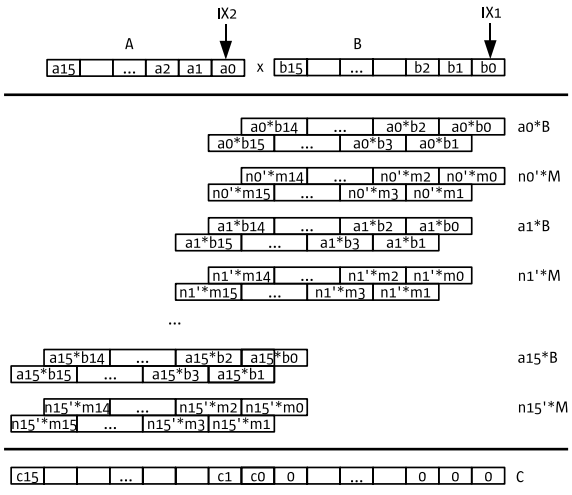

Figure 3. Montgomery modular multiplication.

Fig. 3 depicts the calculation of the Montgomery multiplication and how it is broken up into multiply-accumulate instructions. The given example assumes 1024-bit operands A and B. Operands are broken up into 64-bit words a_i and b_i, $i = 0..15$. The modular product C is the sum of the partial products $a_i * B$ and the reduction terms $n'_i * M$. Not shown is the calculation of $n'_i = (-(1/M) * c_i) \ mod \ t^{64}, c_i = ((a_i * B * t^{64*i} + \sum_{j=0}^{i-1} a_j * B * t^{64*j} + n_j * M * t^{64*j}) * t^{-64*i}) \ mod \ t^{64}$.

The code to compute $C(t) \leftarrow (A(t) * B(t) + C(t)) * t^{-m} \ mod \ M(t)$ over $GF(2^m)$ is given in Table 2[5]. Each line stands for an instruction tuple. Operands are pointers to memory variables. A is the multiplier, B is the multiplicand, M is the modulus, C is the product, and $N' = -(1/M) \mod t^{64}$ and $n' = N' * C_0$ are needed for the reduction computation. The code for multiplication over $GF(p)$ is slightly more complicated as it also has to consider possible carry bits.

$IX1$ and $IX2$ are the indexes used to address the individual words of the multiple-precision operands. Line 0 initializes $IX2$ to zero. Line 1 computes the first word of a partial product and initializes $IX1$ to zero. Line 2 implements a loop that calculates the remaining words of the partial products. The extended carry bits are propagated in that the extended carry of the previous multiplication is added in and the high word of the result is stored in the extended carry. Line 3 is needed to output the extended carry of the last multiplication - 0 refers to a value (and not an address). Lines 1 to 3 also perform the

[5] $C(t) \leftarrow (A(t) * B(t)) * t^{-m} \ mod \ M$ can be calculated by initializing $C(t) = 0$.

Table 2. Montgomery modular multiplication code.

	Arithmetic Instruction	Control Instruction
0	NOPA	CLR IX2
1	MULACCX A+IX2,B,C,C	CLR IX1
2	MULACCXC A+IX2,B+1+IX1,C+1+IX1,C+1+IX1	INCBR IX1, 2
3	MULACCXC 0,0,0,C+1+IX1	NOPC
4	MULACCX N′,C,0,n′	NOPC
5	MULACCX n′,M,C,C	CLR IX1
6	MULACCXC n′,M+1+IX1,C+1+IX1,C+IX1	INCBR IX1, 6
7	MULACCXC 0,0,C+1+IX1,C+IX1	INCBR IX2, 1

	Description
0	IX2 ← 0;
1	Mem[C] ← (Mem[A+IX2] ⊗ Mem[B]) ⊕ Mem[C]; IX1 ← 0;
2	Mem[C+1+IX1] ← (Mem[A+IX2] ⊗ Mem[B+1+IX1]) ⊕ Mem[C+1+IX1] ⊕ XC; if IX1 ≠ RC.IX_MAX then branch to line 2; IX1 ← IX1+1;
3	Mem[C+1+IX1] ← XC;
4	Mem[n′] ← Mem[N′] ⊗ Mem[C];
5	Mem[C] ← (Mem[n′] ⊗ Mem[M]) ⊕ Mem[C]; IX1 ← 0;
6	Mem[C+IX1] ← (Mem[n′] ⊗ Mem[M+1+IX1]) ⊕ Mem[C+1+IX1] ⊕ XC; if IX1 ≠ RC.IX_MAX then branch to line 6; IX1 ← IX1+1;
7	Mem[C+IX1] ← Mem[C+1+IX1] ⊕ XC; if IX2 ≠ RC.IX_MAX then branch to line 1; IX2 ← IX2+1;

addition needed to add the partial result to the sum of the previously accumulated partial products. Line 4 computes the reduction factor n'. Lines 5 to 7 are similar to lines 1 to 3 in that a multiple-precision multiplication is performed, this time, multiplying n' and the modulus M. By adding $n' * M$, the previous result in C is reduced by zeroing out the least significant word of C.

5 A Multiple-Precision Multiplier for $GF(p)$ and $GF(2^m)$

Parallel multipliers typically use a carry-save adder (CSA) tree together with a carry-propagate adder (CPA). The CSA tree calculates the sum of the partial products in a redundant carry/sum representation and the CPA performs the final addition of the carry and sum bits. We modified the CSA tree such that it generates a so-called XOR product in addition to the integer product. The former is needed for ECC over fields $GF(2^m)$ and the latter for RSA and ECC over fields $GF(p)$.

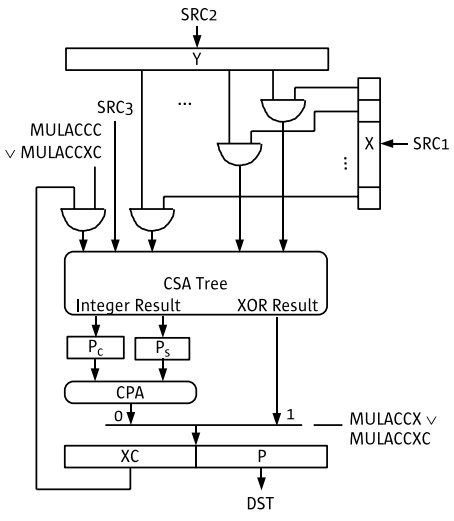

Figure 4. Multiplier.

Figure 4 shows the organization of the multiplier. Registers X and Y hold the multiplicand and the multiplier, respectively. The partial products $X_i * Y, i = 0..n-1$ are added by the CSA and the carry and sum results are stored in registers P_c and P_s, respectively. The sum of P_c and P_s is computed by the CPA and stored in registers P and XC with P holding the low word and XC the high word of the result.

A CSA tree consists of full adder (FA) elements and half adder (HA) elements. In its simplest form, such a tree uses $2n$ chains each consisting of 1 to n FAs and HAs to sum up n partial products. There are techniques to reduce or compress the chain lengths thereby reducing the logic delay to obtain the carry/sum result. With these techniques the tree height is reduced from n to $log_{1.5}n$ [4, 27].

We will now explain how the CSA tree can be modified to obtain the XOR result in addition to the integer result. Looking at the functions realized by the FAs and the HAs we notice that the sum S already provides the XOR function needed: $(FA)\ S = A \oplus B \oplus C_{in}, C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in}; (HA)\ S = A \oplus B, C_{out} = A \cdot B$.

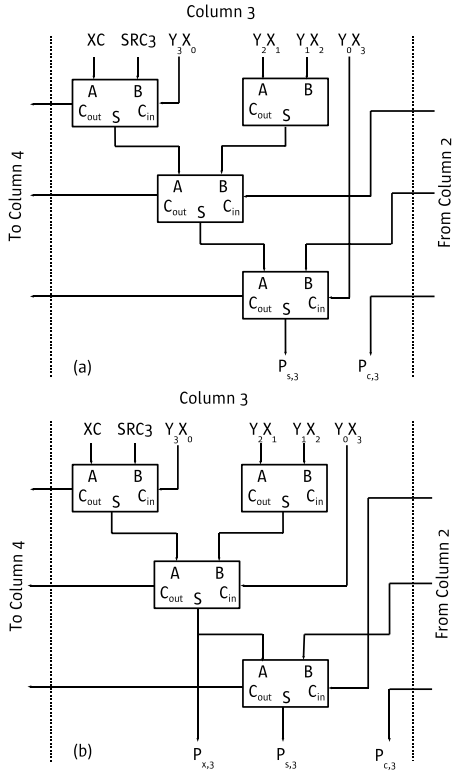

Figure 5. Conventional (a) and modified (b) CSA tree column.

Thus, the XOR result can be obtained by chaining the FAs and the HAs in such a way that the inputs to the CSA tree are added first and the carry bits C_{out} are added as late as possible at the bottom of the tree. Figure 5 illustrates how a column of a CSA tree for a 6x6 multiplier is modified such that an XOR result is generated in addition to an integer result.

The modifications shown require little extra circuitry - some columns require the addition of an XOR gate - and do not increase logic delays[6]. Analyzing the extra cost of adding support for XOR results, we found an average increase of 5% in terms of logic required. It is worth pointing out that the outlined modifications can be easily applied to multiplier designs found in general-purpose processors.

We optimized the multiplier for multiple-precision operations by implementing a multiply-accumulate instruction that combines a multiplication step and an accumulation step. The multiply-accumulate instruction comes in four versions. MULACCC inputs and adds the extended carry $MUL.XC$ whereas MULACC ignores $MUL.XC$. Additionally, there are two sets of instructions, one defined for $GF(p)$ and another one defined for $GF(2^m)$.

Here we want to consider the instruction MULACCC as defined in Table 1. A MULACCC instruction corresponds to one 64-bit multiplication and two 128-bit additions[7]. As illustrated in Fig. 4, the two additions are implemented with the help of the CSA tree. MULACCC generates a 128-bit result whereby the low word is stored in the destination operand DST and the high word is stored in an extended carry register XC that is local to the multiplier. XC is a carry input to the next multiply-accumulate (with carry) instruction.

6 Performance Analysis

Figure 6 gives the distribution of instructions executed for the RSA and ECC algorithms. We chose RSA key sizes of 1024 and 2048 bits, the former representing the key size currently

[6]Since the XOR result is generated early, the critical path does not get affected even if the extra XOR gate is needed. Also, the multiplexer needed to choose between the integer result and the XOR result should not affect the critical path typically given by the CSA tree.

[7]128-bit additions are needed since the result of the multiplication is 128 bits wide.

considered to provide enough security for most applications and the latter offering the next higher level of security. The instruction counts for RSA-1024 and RSA-2048 are compared with the ECC counterparts offering a similar level of security. That is, RSA-1024 is comparable in security strength with ECC-160p and ECC-163b, and RSA-2048 is comparable with ECC-224p and ECC-233b, with p referring to prime integer fields $GF(p)$ and b to binary polynomial fields $GF(2^m)$. For each algorithm, we show the total number of arithmetic instructions executed and the total number of multiply-accumulate instructions executed. The graphs clearly show that both the RSA and ECC algorithms are dominated by multiply-accumulate instructions.

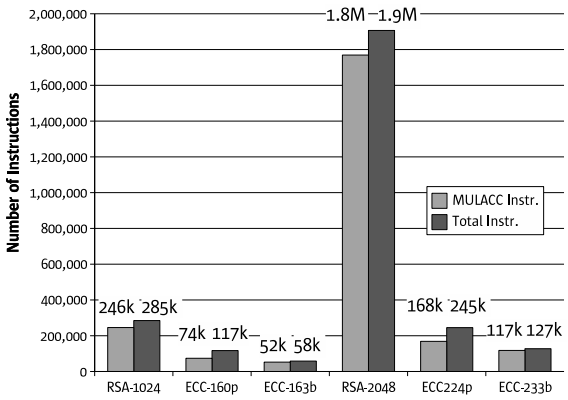

Figure 6. Instruction counts.

Next, we analyze performance. We give projected numbers for a fully pipelined processor running at a 1.5 GHz clock frequency which is representative of the clock speeds of state-of-the-art general-purpose processors. Table 3 lists the number of operations per second for RSA[8] and ECC. These numbers are derived from the instruction counts of Fig. 6 by assuming a fully pipelined data path with a throughput of one instruction per clock. The numbers show a clear performance advantage of ECC over RSA. At present security levels, ECC-160p offers a speedup of 2.4 and ECC-163b a speedup of 4.9 over RSA-1024. At future security levels, the comparison favors ECC even more: ECC-224p is 7.8 times and ECC-233b is 15.0 times faster than RSA-2048. These numbers not only show a significant performance advantage for ECC, they further illustrate that the ratio of the computation times for RSA and ECC is going to dramatically increase as higher security is needed.

Table 3. Performance at 1.5 GHz.

Algorithm	Instr. Count	Op./s	Speedup
RSA-1024	284,900	5,265	1.0x
ECC-160p	117,032	12,817	2.4x
ECC-163b	58,236	25,756	4.9x
ECC-163b-opt	**38,009**	**39,464**	**7.5x**
RSA-2048	1,906,884	787	1.0x
ECC-224p	245,330	6,114	7.8x
ECC-233b	127,365	11,772	15.0x

These performance numbers are somewhat optimistic as data dependencies might lead to pipeline stalls. Refering to the code in Table 2 the data dependencies in the two loops given by lines 1/2 and 5/6, respectively, will cause stalls if the order of the computations is not rearranged. Code execution can be reordered, for example, by interleaving the calculation of $a0 * B, n0' * M, a1 * B, n1' * M, ...$

The performance numbers discussed so far were obtained by running code that uses Montgomery reduction for the modular multiplication. That is, the modulus is not hard-coded; rather, it can be an arbitrary prime number or irreducible polynomial that can be passed to the code as a parameter. It is possible to optimize reduction for a given modulus. This is particularly attractive if the modulus is a pseudo-Mersenne prime, a trinomial, or a pentanomial containing only few terms [2, 26]. Whereas the reduction using a generic

[8] We only consider RSA private-key operations.

modulus corresponds to a full multiple-precision multiplication, optimized reduction for these special moduli can be implemented by a small number of additions, substractions, and shift operations [26]. Since our architecture is microprogrammable, we were able to implement these optimizations simply by modifying the firmware. This is in contrast to other architectures [25], that hardwire the algorithm making it difficult to make changes once the design is completed. To illustrate this with an example, we have listed performance numbers in Table 3 obtained for such an optimized implementation for SECG curve ECC-163b. This implementation also makes use of a technique called partial reduction that we introduced in [5]. The optimizations result in a speedup of 1.5x when comparing generic code (ECC-163b) with curve-optimized code (ECC-163-opt).

7 Conclusions

We have shown that hardware acceleration of public-key algorithms can be added to a general-purpose processor with minimal modifications. To support the emerging elliptic curve cryptosystem in addition to the traditional RSA cryptosystem, a dual-field multiplier is needed that supports operations for both fields $GF(p)$ and fields $GF(2^m)$. We have shown that such support can be provided by a standard integer multiplier simply by rearranging the carry-save adder tree. The resulting modifications do not add any gate delay to the critical path of the multiplier and only require a modest amount of additional chip resources.

An analysis of the distribution of the instructions executed by the ECC and RSA algorithms shows that multiplication is the single-most critical operation. For this reason, we introduced a multiply-accumulate instruction that allows for efficiently scheduling the multiplier. With a single multiply-accumulate instruction, three tasks are being handled: (i) word by word multiplication, (ii) carry propagation, and (iii) partial product accumulation.

The performance analysis shows a clear performance advantage for ECC over RSA. At current security levels, we observe a speedup of 2.4x and 4.9x for ECC $GF(p)$ and $GF(2^m)$, respectively, over RSA. And for future security levels, the corresponding speedups are 7.8x and 15.0x, respectively.

References

[1] ANSI X9.63, *Elliptic Curve Digital Signature Algorithm (ECDSA)*, American Bankers Association, 1999.

[2] Certicom Research, *SEC 2: Recommended Elliptic Curve Domain Parameters*, Standards for Efficient Cryptography, Version 1.0, September 2000.

[3] H. Cohen, A. Miyaji, T. Ono, *Efficient Elliptic Curve Exponentiation using Mixed Coordinates*, Int. Conference on the Theory and Application of Cryptology, ASIACRYPT '98, Advances in Cryptology, Springer Verlag, 1998, pp. 51-65.

[4] L. Dadda, *Some Schemes for Parallel Multipliers*, Alta Frequenza, vol. 34, 1965, pp. 349-356.

[5] H. Eberle, N. Gura, S. Chang Shantz, *Generic Implementations of Elliptic Curve Cryptography using Partial Reduction*, Proc. 9th ACM Conference on Computers and Communications Security, November 18-22, 2002, Washington, DC, pp. 108-116.

[6] H. Eberle, N. Gura, S. Chang Shantz, *A Cryptographic Processor for Arbitrary Elliptic Curves over* $GF(2^m)$, Proc. IEEE 14th Int. Conference on Application-specific Systems, Architectures and Processors, June 24-26, 2003, The Hague, The Netherlands, pp. 444-454.

[7] H. Eberle, N. Gura, S. Chang Shantz, V. Gupta, *A Cryptographic Processor for Arbitrary Elliptic Curves over* $GF(2^m)$, Sun Microsystems Technical Report TR-2003-123, May 2003, extended version of [6].

[8] J. Goodman, A. Chandrakasan, *An Energy-Efficient Reconfigurable Public-Key Cryptography Processor*, IEEE Journal of Solid-State Circuits, vol. 36, no. 11, November 2001, pp. 1808-1820.

[9] V. Gupta, D. Stebila, S. Fung, S. Chang Shantz, N. Gura, H. Eberle, *Speeding up Secure Web Transactions Using Elliptic Curve Cryptography*, 11th Network and Distributed System Security Symposium, February 5-6, 2004, San Diego, CA, pp. 231-239.

[10] V. Gupta, D. Stebila, S. Chang Shantz, *Integrating Elliptic Curve Cryptography into the Web's Security Infrastructure*, 13th World Wide Web Conference – Alternate Track Papers and Posters, May 17-22, 2004, New York, NY, pp 402-403.

[11] N. Gura, H. Eberle, S. Chang Shantz, D. Finchelstein, S. Gupta, V. Gupta, D. Stebila, *An End-to-End Systems Approach to Elliptic Curve Cryptography*, 4th Int. Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002, Lecture Notes in Computer Science 2523, Springer-Verlag, 2002, Redwood Shores, CA, August 13-15, 2002, pp. 349-365.

[12] A. Gutub, M. Ibrahim, *High Radix Parallel Architecture for GF(p) Elliptic Curve Processor*, IEEE Int. Conference on Acoustics, Speech, and Signal Processing, ICASSP 2003, April 6-10, 2003, pp. II - 625-628.

[13] M. Hasan, A. Wassal, *VLSI Algorithms, Architectures, and Implementation of a Versatile $GF(2^m)$ Processor*, IEEE Transactions on Computers, vol. 49 , no. 10 , Oct. 2000, pp. 1064 - 1073.

[14] B. Kaliski, *TWIRL and RSA Key Size*, Technical Note, Revised May 6, 2003, RSA Laboratories, www.rsasecurity.com.

[15] P. Leong, I. Leung, *A Microcoded Elliptic Curve Processor using FPGA Technology*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 10, no. 5, Oct. 2002, pp. 550-559.

[16] K. Leung, K. Ma, W. Wong, P. Leong, *FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor*, IEEE Symposium on Field-Programmable Custom Computing Machines, April 17-19, 2000, pp. 68 - 76.

[17] J. López, R. Dahab, *Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation*, 1st Int. Workshop on Cryptographic Hardware and Embedded Systems, CHES 1999, Lecture Notes in Computer Science 1717, Springer-Verlag 1999, Worcester, MA, August 12-13, 1999 pp. 316-327.

[18] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[19] P. Montgomery, *Modular Multiplication without Trial Division. Mathematics of Computation*, vol. 44, no. 170, April 1985, pp. 519-521.

[20] Netscape Security Services. http://www.mozilla.org/projects/security/pki/nss/.

[21] OpenSSL Project. http://www.openssl.org/.

[22] S. Ors, L. Batina, B. Preneel, J. Vandewalle, *Hardware Implementation of an Elliptic Curve Processor over GF(p)*, IEEE 14th Int. Conference on Application-specific Systems, Architectures and Processors, June 24-26, 2003, The Hague, The Netherlands, pp. 433-443.

[23] G. Orlando, C. Paar, C., *A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$*, 2nd Int. Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000, Springer-Verlag, Lecture Notes in Computer Science 1965, 2000, Worcester, MA, August 17-18, 2000, pp. 41-56.

[24] J. Quisquater, C. Couvreur, *Fast Decipherment Algorithm for RSA Public-key Cryptosystem*, Electronics Letters, vol. 18, no. 21, 1982, pp. 905-907.

[25] A. Satoh, K. Takano, *A Scalable Dual-Field Elliptic Curve Cryptographic Processor*, IEEE Transactions on Computers, vol. 52, no. 4, April 2003, pp. 449-460.

[26] U.S. Department of Commerce, National Institute of Standards and Technology, *Digital Signature Standard (DSS)*, Federal Information Processing Standards Publication FIPS PUB 186-2, January 2000.

[27] C. Wallace, *A Suggestion for a Fast Multiplier*, IEEE Transaction on Electronic Computers, vol. 13, 1960, pp. 14-17.