Journal of Systems Architecture 60 (2014) 440–451

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Improving the computational efficiency of modular operations for embedded systems

Ismail San*, Nuray At

Department of Electrical and Electronics Engineering, Anadolu University, 26470 Eskişehir, Turkey

ARTICLE INFO

Article history:
Received 31 January 2013
Received in revised form 8 September 2013
Accepted 18 October 2013
Available online 31 October 2013

Keywords:
Public-key cryptography
Modular arithmetic
Acceleration
FPGA
System-on-chip

ABSTRACT

Security protocols such as IPSec, SSL and VPNs used in many communication systems employ various cryptographic algorithms in order to protect the data from malicious attacks. Thanks to public-key cryptography, a public channel which is exposed to security risks can be used for secure communication in such protocols without needing to agree on a shared key at the beginning of the communication. Public-key cryptosystems such as RSA, Rabin and ElGamal cryptosystems are used for various security services such as key exchange and key distribution between communicating nodes and many authentication protocols. Such public-key cryptosystems usually depend on modular arithmetic operations including modular multiplication and exponentiation. These mathematical operations are computationally intensive and fundamental arithmetic operations which are intensively used in many fields including cryptography, number theory, finite field arithmetic, and so on. This paper is devoted to the analysis of modular arithmetic operations and the improvement of the computation of modular multiplication and exponentiation from hardware design perspective based on FPGA. Two of the well-known algorithms namely Montgomery modular multiplication and Karatsuba algorithms are exploited together within our high-speed pipelined hardware architecture. Our proposed design presents an efficient solution for a range of applications where area and performance are both important. The proposed coprocessor offers scalability which means that it supports different security levels with a cost of performance. We also build a system-on-chip design using Xilinx's latest Zynq-7000 family extensible processing platform to show how our proposed design improve the processing time of modular arithmetic operations for embedded systems.

1. Introduction

With the advancement of communication technology and information systems, networking and data streaming applications call for higher data rates as well as security at the same time. There are many such emerging applications especially for embedded systems that need to communicate, store or manipulate confidential data. This makes security is a primary concern in the design of embedded systems for certain applications. Security is provided by the set of cryptographic algorithms which are usually computationally intensive algorithms. Hence, embedded security is a constantly developing field along with the research on efficient hardware design of cryptographic algorithms. To meet security needs of such applications, there are several security protocols such as IPSec, SSL and VPNs which are used between a pair of communicating hosts called sender and recipient. Further, with the introduction of Public-Key Cryptography (PKC), a public channel can be used for secure communication. PKC have also many benefits in such protocols such as key-distribution and various authentication protocols due to the fact that PKC does not require a secure initial key exchange between the sender and the recipient.

Most of the public-key cryptosystems use modular arithmetic operations, specifically, modular multiplication and exponentiation. These two operations are also widely used in other fields. Hence, their efficient computation is quite important for the security of embedded systems. There are many different algorithms and computation models in the literature to perform modular multiplication such as Montgomery, Booths, Karatsuba, etc. Moreover, repetitive use of modular multiplication is needed in the modular exponentiation operation. Therefore, efficient design of modular multiplication has a great importance in the computational efficiency of modular exponentiation.

In this study, we focus on high-speed pipelined hardware implementation of the modular multiplication and exponentiation operations on FPGAs. In [1], San and At designed a compact coprocessor which efficiently exploits intrinsic properties of Karatsuba algorithm on Xilinx Virtex-5 FPGA devices. Here, in particular, the design strategy in [1] is improved in terms of latency. Improved pipelined version of Karatsuba coprocessors employed in the

* Corresponding author. Tel.: +90 222 321 3550/6469.
E-mail addresses: isan@anadolu.edu.tr (I. San), nat@anadolu.edu.tr (N. At).

1383-7621/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.sysarc.2013.10.013

proposed architecture in this study. Hence, a pipelined hardware implementation for the modular multiplication and exponentiation operations using both Karatsuba and Montgomery algorithms is proposed.

Today's FPGAs have enhanced hard cores such as DSP and Block RAM components yielding very high operating frequencies. As emphasized by Güneysu, "the use of device specific components lead to considerably higher system performance" [2]. With this approach in mind, we have adapted Karatsuba and high-radix Montgomery modular multiplication algorithms to efficiently compute the modular multiplication on FPGA using device specific components such as DSP and shift register components. In order to extend arithmetic precision to be used in larger modulus and make modular multiplication algorithm more compact for larger bits, Karatsuba and Montgomery algorithms can be used by exploiting the embedding multiplier modules existing in almost all FPGAs in pipelined mode. In this article, we first efficiently combine very high-radix Montgomery and Karatsuba algorithms for performing modular multiplication algorithm using these embedded hard cores on latest high-speed FPGAs. This gives us very powerful results in those cases where the parallel and pipelined processing is available. Then, we implement compact modular exponentiation architecture using the proposed high-speed modular multiplication blocks. We provide hardware performance figures of our methods on FPGA in terms of latency, area, and frequency. We achieve efficient high-radix modular multiplications by means of pipelined Karatsuba coprocessor. The proposed hardware uses efficiently Karatsuba algorithm to multiply large numbers in a compact manner with saving multiplier blocks. We also analyze the effects of radix changes on our hardware architecture. The comparison of our hardware performance results with the results of other reported hardware architectures for modular multiplication and exponentiation is also given. The results show that the architectures proposed in this study for modular multiplication and exponentiation yield good performance with a reasonable area in hardware for embedded systems where FPGA comes into play.

The main contribution of this paper is to present a high-speed hardware architecture for modular multiplication and exponentiation using both Karatsuba and Montgomery modular multiplication algorithms with the presented design methods using embedded building blocks on the latest Xilinx FPGA. The proposed method efficiently exploit embedded multipliers and adder units on FPGA. The presented architecture aims to provide an efficient architecture to be used where modular arithmetic is definitely required such as Coding theory, Cryptography, especially PKC, DSP and so on. There are many extensive studies in the literature related to increase the efficiency of multiplication [3–7], especially in PKC. From our point of view, the main advantage of this method over other existing methods is that iterative utilization of hardware resources with pipelining and tight scheduling brings better performance with smaller logic area. Our method also attains very high frequency. The other advantage of our architecture is its scalability which respect to the operand size.

Two levels of scalability are considered:

- The design should require hardware resources as small as possible.
- The design allows multiplication on larger size on the same architecture with small modifications.

In this study, we also propose a system-on-chip (SoC) design which uses our efficient modular arithmetic hardware architecture as a coprocessor with Xilinx's latest Zynq-7000 family extensible processing platform. We adapt Hardware/Software codesign approach in order to exploit the advantages of both methodologies. The design rationale of the proposed architecture is to accelerate the performance of computational intensive tasks required by the modular arithmetic operations. Furthermore, the low latency presented by the coprocessor empowers to operate in higher frequencies within the SoC platforms.

The paper is outlined as follows. Section 2 briefly overviews the literature on the algorithms for modular multiplication and exponentiation operations and their realizations on FPGA devices. After we give the key preliminaries for this study in Section 3, we present our modular arithmetic coprocessor including all hardware aspects in Section 4. We then illustrate our SoC design, which uses our modular arithmetic coprocessor, including the details for the Zynq-7000 based embedded system and its communication mechanism with the coprocessor. In Section 6, we show the performance results of the proposed SoC architecture and discuss our design from all perspectives. Finally, some conclusions are drawn in Section 7.

2. Previous work

There are many studies about efficient implementations of modular arithmetic operations included multiplication and exponentiation using various techniques to decrease the computation of these operations, especially in public key cryptography. Montgomery [8] and Karatsuba [9,10] multiplication algorithms are the most efficient and popular algorithms.

High-radix algorithms [7,11] have been proposed for improving the performance. However, as the radix increases, the design complexity and the length of clock cycle also increase dramatically due to requiring the use of larger digit multipliers. These high-radix designs generally consumes huge amounts of hardware area. So, previously, low-radix designs are more attractive for hardware implementation due to the lack of larger digit multipliers. However, dedicated 17-bit embedded multipliers can easily be found in almost all FPGAs today and implementing larger than 17-bit multiplier is also possible with Karatsuba algorithm. By efficiently exploiting Karatsuba algorithm, one can achieve a larger bit multiplier by using less multiplier units. By using Karatsuba algorithm, doubling the bit-width of the multiplier is achieved in less amount of multiplier compared to classical multiplication. Employing Karatsuba algorithm in a pipelined mode allows to perform high-radix Montgomery modular multiplication based on these Karatsuba coprocessors within much less clock cycles. Our aim is to find a compact trade-off between the computation time and the required hardware area by efficiently exploiting Montgomery and Karatsuba algorithms together. Analysis of the design trade-offs for high-radix modular multipliers is found in [12].

Tenca and Koç proposed a radix-2 scalable Montgomery multiplication architecture [13] which multiplicand is scanned word-by-word and the other operand, multiplier, is scanned bit-by-bit. This multiple word radix-2 Montgomery multiplication allows efficient scalable hardware implementation. Parallelism among instructions of the algorithm in different scanning bits of multiplier is possible. The main difference and advantage of the architecture compared to the other algorithms in the literature is its scalability to any operand size which in fact enables to find good design trade-offs for different application needs. Tenca et al. [14] describe a scalable Montgomery multiplier which is adjustable to constrained areas and capable to work on any given precision of the operands. The algorithm proposed in [14] uses Booth encoding technique and a radix-8 scalable multiplier is implemented to show the performance of the design. In contrast to [14], our proposed hardware architectures operate with very high-radix values to reveal the computational efficieny of very high-radix changes in the modular multiplication and exponentiation.

The study [15] proposes an automatic generation of modular multipliers especially for FPGA based systems. Beuchat and Muller [15] present an algorithm which can select best possible high-radix carry-save representation for predetermined modulus. A synthesizable VHDL code for given parameters is automatically generated by the method proposed in [15].

Multiplication of large numbers is an important arithmetic operation in polynomial multiplication for signal processing, coding theory [16]. For a comprehensive survey of different algorithms to perform polynomial multiplication, we refer the reader two studies Bernstein [17] and Nedjah et al. [18]. In order to evaluate the complexity of an algorithm, computational efficiency, memory requirement and resource sharing offered by the algorithm have great importance. There are many other studies on the implementations of large digit multiplication operation using various methods. We select Karatsuba multiplication algorithm [9] since it is well-suited to FPGA based environment.

Dinechin et al. [3] aim to build large multiplier by using fewer DSP48E blocks. The proposed architectures in [3] achieve very high frequencies since the design consumes few additional logic that have little impact on the critical path of the architecture. The presented architectures [3] in for different bit width multiplication differs from each other. However, our architecture is same for different multiplication sizes. Our architecture is generic and scalable. In addition, our coprocessor efficiently exploits DSP48E1 blocks and requires a small amount of logic. Suzuki [19] presents a method that can exhibit the the maximum performance of available FPGA resources for modular exponentiation operation. The study exploits the embedded building blocks including DSP48E, Block RAM and SRL16 to maximize the utilization of FPGA resources. Due to the need to the store preliminary input X_{2i+1} in [19], Block RAM utilization is required and the number of Block RAM changes with different parameters of the design. Gao et al. [20] proposes FPGA-based efficient large size signed multipliers using multigranular small embedded blocks to be used in the applications such as scientific computation, cryptography, and data intensive systems. Our proposed method differs from this study since our proposed hardware architecture employs a powerful pipelining mechanism with a very tight scheduling using DSP48E1 blocks and a small amount of configurable logic slices. We compare this study with our coprocessor results in Section 6. Chow et al. [21] presented a Montgomery multiplier that exploits Karatsuba algorithm for cryptography applications. The design uses multiple precision arithmetic techniques in order to make the critical path delay independent to bit width of the multiplier. They recursively implement Karatsuba multiplier to achieve n-bit multiplier to use in Montgomery algorithm in one pass. In contrast to that study, we implement pipelined Karatsuba multiplier in an iterative manner together with very high-radix Montgomery multiplier in this study.

Bo et al. [6] presents a hardware architecture for RSA encryption based on Montgomery modular multiplication. They propose an hardware architecture which is able to perform necessary operations for the RSA encryption using only one DSP48E1, one Block RAM and a few slices. Moreover, the DSP48E1 block works almost full which yields an efficient solution for low-cost applications. Their proposed architecture is well-suited to many applications where resources are constrained. In [22], Bo et al. aims to accelerate the architecture presented in [6] using CRT technique. There is another recent study in designing coprocessor for Modular arithmetic operations. Bautista et al. [23] recently proposes a mathematical coprocessor which is able to perform modular arithmetic operations based on FPGA. Their proposed design is based on Montgomery algorithm for modular multiplication.

Our motivation of this study is to find a good trade-off between the area and the computation time of modular multiplication and exponentiation operations for hardware implementations. For this purpose, we propose a pipelined method which utilizes both Karatsuba and Montgomery modular multiplication algorithms by using embedded DSP blocks in FPGA to obtain efficient hardware design. We provide the performance figures of our design in Tables 1, 2 and 4. The comparisons with respect to other reported studies in the literature are given in Table 3.

3. Preliminaries

In this section, we introduce the basic notations and background of the proposed method. We first review the Karatsuba algorithm that we utilize for high-radix parallel multiplier in this study. Then, we briefly introduce the Montgomery modular multiplication and exponentiation algorithms needed for performing modular arithmetic. In order to construct a PKC based on modular arithmetic, we use Montgomery reduction and right-to-left modular exponentiation algorithms, which are the standard algorithms for modular multiplication and exponentiation.

3.1. Karatsuba algorithm

The Karatsuba algorithm was introduced by Karatsuba and Ofman for multiplication of large integers [9]. The algorithm reduces the overall running time of the multiplication of two N-digit integers to $\mathcal{O}(N^{\log_2^3})$, as compared to $\mathcal{O}(N^2)$ in the classical algorithm. One multiplication is replaced with a three addition or subtraction operations. It is thus faster than the classical multiplication algorithm. Karatsuba algorithm reduces the number of single precision multiplications by taking advantage of two intermediate partial products.

To illustrate the Karatsuba algorithm, let X and Y be two $2k$-bit unsigned integers. One can split them as follows

$$X = 2^k \cdot X_1 + X_0 \text{ and } Y = 2^k \cdot Y_1 + Y_0$$

Four k-bit multiplication and three addition operations are required to compute the product $X \cdot Y$ in classical long multiplication operation which is illustrated in Eq. 1.

Table 1
Synthesis results for the proposed modular multiplication coprocessor on Virtex-7 FPGA.

FPGA	n	Radix size [w]	Area [slices]	DSP48E1 blocks	Frequency [MHz]	Processing time [μs]
Virtex-7 (7vx330t-3)	512	16	128	7	478	2.29
		32	382	13	288	1.02
	1024	16	170	7	479	8.83
		32	536	13	490	2.24
		64	1257	27	211	1.39
	2048	16	235	7	479	34.75
		32	567	13	490	8.64
		64	1785	27	403	2.73

Table 2
Synthesis results for the proposed modular exponentiation coprocessor on Virtex-7 FPGA.

FPGA	n	Radix size [w]	Area [slices]	DSP48E1 blocks	Frequency [MHz]	Processing time [ms]
	512	16	343	14	458	1.23
		32	801	26	283	0.54
		16	385	14	468	9.28
Virtex-7 (7vx330t-3)	1024	32	1060	26	485	2.33
		64	3046	54	201	1.52
		16	553	14	370	92.25
	2048	32	1092	26	413	21.03
		64	3558	54	399	5.68

Table 3
Performance results of hardware architectures for modular exponentiation on FPGAs.

	Algorithm	FPGA	n	Area [slices]	18K-bit RAM	DSP blocks	Frequency [MHz]	Processing time [ms]
	Redundant	XC2VP30–6	512	5868	15	32 18-bit Mul.	52.57	0.645
			1024	11589	29	64 18-bit Mul.	52.9	2.521
Nakano et al. [26]	Non-Redundant	XC2VP30–6	512	3911	15	31 18-bit Mul.	30.40	1.113
			1024	7708	29	61 18-bit Mul.	18.46	7.218
Tang et al. [27]	Using Parallel	XC2V3000-6	512	8235	–	32 18-bit Mul.	99.26	0.59 †
	Multipliers		1024	14334	–	62 18-bit Mul.	90.11	2.33 †
Blum and Paar [7]	High-radix	XC40250XV-9	512	3413 CLBs	–	–	–	2.93
	Montgomery		1024	6633 CLBs	–	–	–	11.95
Suzuki [19]	High-radix	XC4VFX12–10	512	4190	7	17 DSP48	200–400	0.261
	Montgomery		1024	4190	7	17 DSP48	200–400	1.71
	High-radix		512	801	–	26 DSP48E1	283	0.54
This work	Montgomery and	7VX330T-3	1024	3046	–	54 DSP48E1	201	1.52
	Karatsuba		2048	3558	–	54 DSP48E1	399	5.68
	High-radix		512	180	1 36K-bit	1 DSP48E1	447	4.99
Bo et al. [6]	Montgomery	XC6VLX240T-1	1024	180	1 36K-bit	1 DSP48E1	447	36.37
			2048	180	1 36K-bit	1 DSP48E1	447	277.26
Bo et al. [22]	High-radix	XC6VLX240T-1	1024	201-Slice Register	1 36K-bit	1 DSP48E1	410	11.263
	Montgomery			374-Slice LUTs				

†Average number of modular multiplication is considered.

Table 4
The perfomance figures of various SoC architectures on Zynq-7000 based extensible processing platform.

SoC Architecture	n	Mod. Exp. Coproc.	DDR3 SDRAM	Area [slices]	DSP48E1 blocks	Coprocessor Frequency [MHz]	Processing time [ms]
Zynq-7000 based	512	–	✓	–	–	–	116
Pure Software	1024	–	✓	–	–	–	373
SoC architecture	2048	–	✓	–	–	–	1265
Zynq-7000 based	512	✓	✓	1363	26	100	1.54
Mod. Exp. Coproc.	1024	✓	✓	5876	54	100	3.06
SoC with AXI4-Lite	2048	✓	✓	5945	54	100	22.67
Zynq-7000 based	512	✓	✓	1576	26	100	1.52
Mod. Exp. Coproc.	1024	✓	✓	6152	62	100	3.04
SoCwith AXI4	2048	✓	✓	6224	62	100	22.65

$$\begin{aligned} &X \cdot Y \leftarrow 2^{2k} \cdot M_\beta + 2^k \cdot M_\gamma + M_\alpha \\ &M_\beta \quad \leftarrow X_1 \cdot Y_1, \\ &M_\gamma \quad \leftarrow X_0 \cdot Y_1 + X_1 \cdot Y_0, \\ &M_\alpha \quad \leftarrow X_0 \cdot Y_0 \end{aligned} \tag{1}$$

Karatsuba observed that the middle term M_γ can be computed by using two intermediate partial products, M_α and M_β.

Eq. 2 shows the computation of the middle term in Karatsuba method.

$$\begin{aligned} M_\gamma &\leftarrow X_0 \cdot Y_1 + X_1 \cdot Y_0 \\ &\leftarrow X_1 \cdot Y_1 + X_0 \cdot Y_0 + (X_0 - X_1) \cdot (Y_1 - Y_0) \\ &\leftarrow M_\beta + M_\alpha + K_\alpha \cdot K_\beta \end{aligned} \tag{2}$$

Then, from Eqs. (1) and (2) , the product $X \cdot Y$ is computed with three k-bit multiplications, four additions, and two subtractions. So, the Karatsuba method saves a quarter of the multiplications at the cost of some extra additions and subtractions.

3.2. High-radix Montgomery multiplication algorithm

Many cryptosystems, especially PKC, generally computes $X \cdot Y \mathrm{mod} M$ operation. One of the most efficient algorithm for performing this operation in hardware is the Montgomery algorithm [8]. Algorithm 1 shows the pseudo code for Montgomery multiplication in high-radix-2^k. The radix-2 Montgomery modular multiplication requires less area compared to high radix versions. However, its computational performance is worse than

Montgomery modular multiplication with high-radix values. High-radix brings area cost due to the need for high-radix multipliers. The use of high radix values in Montgomery reduction allow to use hard multiplier blocks in FPGAs. We describe the technique in detail that we used to implement high radix Montgomery modular multiplication algorithm and the other necessary mathematical operations for a public-key cryptosystem.

Algorithm 1. Radix-2^k Montgomery Modular Multiplication

```
Input: $M = (M_{s-1},\dots,M_0)_r, X = (X_{s-1},\dots,X_0)_r, Y = (Y_{s-1},\dots,Y_0)_r$, where
  $0 \leqslant X, Y < M, r = 2^k, s = \lceil \frac{n}{k} \rceil, R = r^s$ with $gcd(M, r) = 1$ and
  $M' = -M^{-1} \bmod r$.
Output: $X \cdot Y \cdot R^{-1} \bmod M$
1: $Z[0] = (Z[0]_{s-1},\dots,Z[0]_0)_r$;                     Initialization
2: for $i \leftarrow 0$ to $s-1$ do
3:    $T[i] \leftarrow (Z[i]_0 + X_0 \cdot Y_i) \cdot M' \bmod r$;      Scanning
4:    $Z[i+1] \leftarrow (Z[i] + X \cdot Y_i + M \cdot T[i])/r$;
5: end for
6: if $Z[s] > M$ then
7:    $Z[s] \leftarrow Z[s] - M$;                              Last reduction
8: end if
9: return $Z[s]$;
```

3.3. Montgomery exponentiation algorithm

Most of PKC are based on heavily modular exponentiation operations. This operation requires huge computational complexity due to performing repetitive modular multiplication. Modular exponentiation and multiplication with very big exponent and modulus (specifically longer than 512-bits) are two of the well-known important arithmetic operations in several modern cryptographic algorithms. It has great importance in modern cryptographic algorithms for authentication, electronic signature, and key exchange.

Efficient computation of the modular exponentiation is very important for PKC. The modular exponentiation can be realized by performing a series of modular squaring and multiplication operations. Hence, it is time-consuming operation in the case of large operand sizes. Our approach to achieve high-throughput design is to decrease the execution time of each modular squaring and multiplication operations. In this article, we mainly propose an architecture for public-key cryptosystem which consists of high-speed modular multiplication module.

One of the most efficient algorithm for performing modular exponentiation in hardware is the binary modular exponentiation algorithm which is also known as square-and-multiply algorithm. We use right-to-left binary modular exponentiation to exploit the parallelism between the squaring and multiplication operations.

Algorithm 2. Right-to-Left binary modular exponentiation

```
Input: Positive integers $B, E, M, -M^{-1}, R^2$ $E = \sum_{i=0}^{E_b-1} E_i \cdot 2^i$,
  $E_i \in \{0,1\}$, $R^2 = 2^{64 \cdot d} \bmod M$
Output: $P = B^E \bmod M = ModExp(B, E, M)$
1: $S_0 \leftarrow ModMult(R^2, 1, M)$;                  Initial phase
2: $Z_0 \leftarrow ModMult(R^2, B, M)$;
3: for $i \leftarrow 0$ to $n-1$ do
4:    if $E_i = 1$ then
5:       $S_{i+1} \leftarrow ModMult(S_i, Z_i, M)$;          Multiply
6:    else
7:       $S_{i+1} \leftarrow S_i$;
8:    end if
9:    $Z_{i+1} \leftarrow ModMult(Z_i, Z_i, M)$;             Square
10: end for
11: $S_{n+1} \leftarrow ModMult(S_n, 1, M)$;              Final phase
12: return $P \leftarrow S_{n+1}$;
```

Algorithm 2 shows the pseudo-code for modular exponentiation that we use in this study. Algorithm 2 requires $2n$ modular multiplication operations in worst case. Since there is no data dependency in between modular squaring and multiplication in this exponentiation algorithm, they can be performed in parallel. Thanks to the parallel execution of modular squaring and multiplication operations, modular exponentiation is completed in $n \cdot \tau_{mm}$ time where τ_{mm} is the time required to complete one modular multiplication operation.

4. Modular arithmetic coprocessor

In this study, our main aim is to propose a high-speed and efficient coprocessor in order to perform modular arithmetic operations efficiently including multiplication and exponentiation. The proposed coprocessor is able to meet the requirements of embedded systems in terms of area and performance. Performing these operations in hardware domain accelerate overall system performance of the application which includes modular arithmetic operations.

There are several design approaches that can be applied to cryptographic algorithms for an efficient hardware implementation such as a low-area coprocessor design approach [24], a compact coprocessor design approach [1], high-speed architecture design approach [2] and other design approaches. The most important optimization goals for the coprocessor proposed in this study are to achieve high frequency values for high-speed performance and to utilize the pipelining technique for increasing the throughput. The description of our design strategies for our hardware architecture described in detail in the following sections to show how the coprocessor achieve the demanding constraints of modular arithmetic operations.

4.1. Design rationale

Modular exponentiation operation in nature consists of performing repetitive modular multiplications. If the size of the modulus is increased, the required number of multiplications are also increased. These loop based repetitions of the multiplication are computed in sequential order. Such iterative modular multiplication operations need to be computed very efficiently to achieve higher performance results. Unfortunately, performing modular multiplication operation using radix-2 Montgomery reduction requires a large amount of time with the advantage of consuming a small amount of area. For the applications where the processing time is more important than the required hardware area, high-radix Montgomery reduction is used. However, high-radix Montgomery reduction requires high-radix multipliers which consumes more area and more latency with respect to lower radix sizes. In this study, our aim is to find good trade-off for high-performance modular arithmetic operations for embedded systems. We look for the various design trade-offs by implementing our hardware architecture in several high-radix sizes. We take advantage of the intrinsic parallelism in the Montgomery high-radix modular multiplication and Karatsuba algorithms by efficiently exploiting embedded DSP building blocks of FPGAs. In summary, our design philosophy for this study consists of following design methods:

- *Pipelining*: we propose a pipelined ALU to achieve maximum clock frequency. We separate the necessary computation into a set of equal stages. We balance each stages by utilizing DSP48E1 blocks in order to achieve high frequency values. Each stage in the ALU processes one part of the algorithm. Our pipelined ALU allows a parallel execution of different parts of the algorithm at the same time. Hence, the stages of our pipelined ALU operates simultaneously using different resources which increases the throughput of the system.

- *Interleaving*: we interleave the independent operations performing modular multiplication algorithm that we used in this study. This allows us to obtain a tight scheduling which increases our throughput substantially.
- *Reusing*: we reuse a modular multiplication component for a series of modular multiplication operation, which cannot be parallelized, in modular exponentiation operation to decrease the resource utilization.

The design philosophy that we follow in this study allows us to develop a scalable and high-speed coprocessor for the modular arithmetic operations. All the presented design strategies significantly improve the performance of computationally intensive modular arithmetic operations.

4.2. Large-digit multiplier based on Karatsuba algorithm

Here, we adapt the Karatsuba algorithm for DSP block in Xilinx FPGA. There is an overhead of long additions required by Karatsuba algorithm which in fact decreases the critical path of hardware implementations. However, our hardware computes these long additions sequentially. Thus, it does not cause to decrease in frequency for our proposed coprocessor. The pseudo code of our adapted Karatsuba algorithm for FPGA which is given in Algorithm 3.

Algorithm 3. Adapted Karatsuba Algorithm

```
Input: $n$ : size of the Karatsuba algorithm
    $X \in \{0,1,\ldots,2^n-1\}$, $Y \in \{0,1,\ldots,2^n-1\}$,
Output: $Z = Karatsuba(X,Y)$, $Z \in \{0,1,\ldots,2^{2\cdot n}-1\}$
1: if $n = 17$ then
2:   $Z \leftarrow X \cdot Y$;                                  17-bit Embedded Multiplier
3: else
4:   $M_\alpha \leftarrow Karatsuba(X_0, Y_0)$;                 Parallel call 1
5:   $M_\beta \leftarrow Karatsuba(X_1, Y_1)$;                  Parallel call 2
6:   $M_\gamma \leftarrow Karatsuba(X_0 - X_1, Y_1 - Y_0)$;     Parallel call 3
7:   $Z[\frac{n}{2}-1:0] \leftarrow M_\alpha[\frac{n}{2}-1:0]$;        Least
8:   $S_\alpha \leftarrow (M_\alpha \gg \frac{n}{2}) + M_\alpha + M_\beta + M_\gamma$;
9:   $Z[n-1:\frac{n}{2}] \leftarrow S_\alpha[\frac{n}{2}-1:0]$;        Middle
10:  $Z[2\cdot n-1:n] \leftarrow (S_\alpha \gg \frac{n}{2}) + M_\beta$;   Most
11: end if
12: return $Z$;
```

This short description of Karatsuba algorithm gives some insights on designing a dedicated coprocessor. San and At [1] designed a compact coprocessor which efficiently exploits intrinsic properties of Karatsuba algorithm on Xilinx Virtex-5 FPGA devices. The design strategy is based on coprocessor approach. The architecture in [1] consists of a register file implemented by means of dual-ported memory, an ALU, and a control unit. The register file is organized into 34-bit words, and stores multiplication operands. In this paper, move one step forward and propose a improved version in terms of latency. In this study, we extended it to come up with an high-performance hardware architecture by exploiting the pipelining mechanism. This allows us to use in high-radix Montgomery modular multiplication algorithm to increase the computational efficiency of modular arithmetic. Presented pipelined hardware implementing long integer multiplication consumes a few amount of DSP blocks existing in FPGAs. It exploits 17-bit multipliers and 48-bit adder units in DSP blocks to compute the multiplication of high-radix integers in parallel. The pipeline mechanism that we add to our Karatsuba coprocessor [1] increases the clock frequency and aids to achieve high throughput.

4.3. High-speed modular multiplication and exponentiation

The architecture of modular multiplication component has a great impact on the performance of the coprocessor since modular exponentiation consists of a series of modular multiplications. When the design techniques which are presented in Section 4.1 are applied to high-radix Montgomery modular multiplication algorithm, we come up with an Algorithm 4 to enhance the computation of the PKC. Algorithm 4 gives our description of radix-2^w Montgomery modular multiplication on Virtex-7 FPGA. We show our pipelining mechanism in time-sliced fashion in the lines of Algorithm 4. Each line of the algorithm is responsible for different stages of the pipeline so that each of them can be executed in parallel.

Algorithm 4. Radix-2^w Montgomery multiplication on our coprocessor

```
Input: radix-$2^w$, $d = \lceil \frac{n}{w} \rceil$, $w \cdot d \geqslant n+3$,
$X, Y, M, S_i \in \{0,1,\ldots,2^n-1\}$, $-M^{-1} \in \{0,1,\ldots,2^w-1\}$,
    $X = \sum_{i=0}^{d-1} 2^{w\cdot i} \cdot X_i, X_i \in \{0,1,\ldots,2^w-1\}, X_d = 0$,
    $Y = \sum_{i=0}^{d-1} 2^{w\cdot i} \cdot Y_i, Y_i \in \{0,1,\ldots,2^w-1\}$,
    $M = \sum_{i=0}^{d-1} 2^{w\cdot i} \cdot M_i, M_i \in \{0,1,\ldots,2^w-1\}, M_d = 0$,
    $S_i = \sum_{j=0}^{d-1} 2^{w\cdot j} \cdot S_{(i,j)}, S_{(i,j)} \in \{0,1,\ldots,2^w-1\}, S_d = 0, \delta_0, \delta_1, \delta_2$
    $\in \{0,1,\ldots,2^{47}-1\}, \delta_{i,high} = \delta_{i,(47:w)}$,
    $\delta_{i,low} = \delta_{i,(w-1:0)}$
Output: $S_d \leftarrow X \cdot Y \cdot 2^{-w\cdot d} \bmod M = ModMult(X,Y,M)$,
1: $S_0 \leftarrow 0$;                                            Initialization
2: for $i \leftarrow 0$ to $d-1$ do
3:   // On-the-fly $q_i$ computation
4:   $\alpha \leftarrow X_0 \cdot Y_i$;
5:   $\beta \leftarrow \alpha + S_{i,0}$;
6:   $q_i \leftarrow \beta \cdot (-M^{-1}) \bmod 2^w$;                $q_i$ computation
7:   // Pipelined computation of next value of S
8:   $\delta_1 \leftarrow [X_0 \cdot Y_i]_{(w-1:0)} + [q_i \cdot M_0]_{(w-1:0)}$;
9:   $\delta_2 \leftarrow \delta_1 + S_{i,0}$;
10:    $S_{(i+1,0)} \leftarrow \delta_{2,low}$;                      First word of $S_{i+1}$
11:    $C \leftarrow \delta_{2,high}$;
12:    for$j \leftarrow 1$ to $d$do
13:      $\delta_0 \leftarrow [X_{j-1} \cdot Y_i]_{(2w-1:w)} + [q_i \cdot M_{j-1}]_{(2w-1:w)}$;
14:      $\delta_1 \leftarrow \delta_0 + [X_j \cdot Y_i]_{(w-1:0)} + [q_i \cdot M_j]_{(w-1:0)}$;
15:      $\delta_2 \leftarrow \delta_1 + S_{i,j} + C$;
16:      $S_{(i+1,j-1)} \leftarrow \delta_{2,low}$;              Remaining words of $S_{i+1}$
17:      $C \leftarrow \delta_{2,high}$;
18:    end for
19: end for
20: if $S_d \geqslant M$ then
21:    $S_{d+1} \leftarrow S_d - M$;                                  Last reduction
22: end if
```

In order to increase the performance of modular exponentiation, we interleave the q_i computation (Algorithm 4, lines 3 and 6) and the computation of next value of S_{i+1} (Algorithm 4, lines 12 and 18). This approach allows us to generate the q_i values on-the-fly. Initially, it is necessary to compute the first q_0 content before the first computation of S_1. It is mandatory since S_1 depends on q_0. We manage to perform this operation during loading the input data to the input shift registers. After initial computing of q_0 content, the q_i computation is synchronized with the S_{i+1} computation.

4.4. Arithmetic and Logic Unit of the coprocessor

Our architecture is mainly built around a w-bit datapath. The ALU performs all the operations required by the algorithms of interest. In Fig. 1, we illustrate our pipelined ALU for the high-radix modular multiplication architecture, *ModMult* function given in Algorithm 4. It requires to design high-radix multipliers in the datapath. On FPGAs, the best design strategy consists in implementing this multiplier by means of DSP48E1 blocks. The control signals allow us to perform the operations defined in the Algorithm 4 very efficiently. Our ALU has several pipeline units to improve the performance of the computation of modular multiplication operation. We manage to keep these pipeline units continuously busy without any pipeline stall. Hence, we substantially improve the performance of the modular operations. Thanks to these pipeline mechanism, data is supplied by FIFO elements which are realized by SRL32 primitives. These FIFO elements for input and output are illustrated in Fig. 1. In this figure, R*i* denotes a *w*-bit register and they stores the output of high-radix multipliers to compute the next value of S (Algorithm 4, line 16). Each stage in the pipeline is responsible for different lines of Algorithm 4. The *w*-bit blocks of the modular multiplication result are stored in R8 sequentially and they are routed to the output shift registers in a serial manner. When the computation of current S value is completed, the output of the shift register provide this value to the adder network to compute next S value.

4.5. Control unit

The control unit consists of a counter mechanism and shift registers. The counter mechanism simply counts the number of iterations required for the size of the exponents of the modular exponentiation. It has three counters: two of them are counting the inner and outer loop of the Algorithm 4, the last one is required to count the exponent size. Shift registers in the control unit are responsible to manage the control signals for the datapath in order to manage the pipeline delays.

The user starts the modular multiplication or exponentiation by loading the input operands into input shift registers and applying a pulse for Start control signal. Then, fully autonomous modular arithmetic coprocessor computes the required output value and assert a Ready signal to indicate that output is ready.

4.6. Scheduling

We achieve very tight scheduling for our pipelined datapath by our control unit. Control unit is relatively simple due to the nature of the process structure of our pipeline. The operations are always performed in sequence. Hence, we just need to count and start new process with a counter mechanism. In this section, we show one frame of the scheduling to demonstrate how we achieve this tight scheduling. Thanks to the pipelining mechanism of Algorithm 4, we manage to keep our ALU full without any pipeline stall. Fig. 2 illustrates our scheduling of our high radix Montgomery multiplication implementation:

- $q_0 \cdot M_0$ and $X_0 \cdot Y_0$ multiplication operations are executed and the most significant *w*-bit content is stored in register R1 and R2, respectively. At the same time, the least significant *w*-bit content is stored in register R4 and R5, respectively.
- Then, we execute the instruction $R3 \leftarrow R1 + R2$ and obtain δ_0 (see Algorithm 4, line 13), the most significant *w*-bit parts of the multiplications are added and provided to next adder with one cycle delay; at the same time, we execute the instruction $R6 \leftarrow R3 + R4 + R5$ and obtain δ_1 (see Algorithm 4, line 14), the least significant *w*-bit parts of the multiplications are added with the addition of the previous most significant *w*-bit parts; we also execute the instruction $R8 \leftarrow R6 + R7 + \text{CarryOut}$ to compute δ_2 (see Algorithm 4, line 15). In this clock cycle, $S_{(0,0)}$ is also loaded in register R7. This synchronization is required for summation corresponding parts of the results. While these instructions are performed on the adder units, new $q_0 \cdot M_1$ and $X_1 \cdot Y_0$ multiplication operations are executed on the multiplier units and the most and the least significant *w*-bit contents are stored in their corresponding registers.
- In each clock cycle, these addition instructions are performed to sum the provided inputs from multipliers without any stall. Addition of the most significant *w*-bit parts of the multiplication results are always stored in R3 and this result is always summed with the least significant *w*-bit of the consequent multiplication results. Here, *n*-bit multiplication is realized in sequence by *w*-bit words. One cycle delay between these two additions is required to match corresponding *w*-bit parts of the *n*-bit multiplication result. The result of the addition of the least significant parts added to previous S content (see Algorithm 4, line 15).
- At the end of the $d + 1$ clock cycles, the whole S value is computed and stored into the output FIFO. d clock cycle is required due to the depth of $d = \lceil \frac{n}{w} \rceil$. One more clock cycle is required for the division operation which in fact exist in the Montgomery modular multiplication algorithm.
- After $d + 1$ clock cycles, new inputs are provided to the multipliers and adders. The core of the algorithm (see Algorithm 4, line 12 and 18) is performed without any stall in these multiplier and adder network.

On-the-fly computation of q_i allows us to implement modular multiplier without any pipeline stall. q_i content is always required in multiplications. Hence, in order to start a new multiplication q_i should be ready. While initial data is loading to input FIFOs, very first q_0 content is started to be computed. When storing the data is completed, the multiplications can be started since q_0 is ready at the input of the multiplier. The core of the Algorithm 4 is started to compute next S_1 value. While this operation is in progress in the multiplier and adder network, next q_1 is calculating on-the-fly computation unit. When the last element of S_1 is ready, the next S_2 value computation can be started without any stall since q_1 is ready on the input of the multiplier. This on-the-fly computation requires an extra 1 DSP48E1 blocks and some control signals. However, it gives great performance improvement in terms of throughput.

There is a limit to have an architecture to compute next S values without any stall. A FIFO is placed in the output of the q_i generation in order to right scheduling. It needs to have a certain delay sizes in compliance with the parameters of the modular multiplier. This FIFO is called q_i pipeline and it is drawn in Fig. 1. The number of stages of this pipeline is calculated by the equation below.

$$\text{The number of stages of } q_i \text{ pipeline} = (d + 1) - [2 \ * (4 + l) + 3]$$

For instance, d is equal to 16 for modulo size 1024 and radix size 64. Using this equation, it is necessary to implement the Karatsuba multiplier with a latency which is equal to 3. It is not allowed to have more latency in the Karatsuba multiplier blocks for these design parameters.

5. System-on-chip design

An application can be considered to show our high-speed coprocessor architecture is well-suited for many embedded systems which needs public key cryptography. In order to evaluate the performance achieved by our coprocessor compared to the pure software solution of modular exponentiation operation, we build three different SoC architectures on Xilinx's latest Zynq-7000 family extensible processing platform. These three different SoC scenarios show the compactness of the algorithm in terms of required area and achieved processing time.

With the rapid improvement in VLSI technology, the number of components that can be placed on a single chip has been continuously rising. This leads various computationally intensive applications to be implemented as a SoC design. A typical SoC consists of processors, peripherals, or application specific coprocessors. In this work, we propose a SoC design that accelerates modular arithmetic operations using the proposed coprocessor.Note that if a higher security level is desired, then more computational power is needed since strong cryptographic algorithms should be implemented in the system. In pure software systems, high level languages such as C ease the integration of the security primitives. We use the Zynq platform in order to include such a level of abstraction. This processor allows performing computationally intensive tasks required by the modular operations in the hardware domain linked by a custom peripheral to the processor. The proposed work employs Hardware/Software codesign methodology harnessing benefits of both development methodologies. The operations required by PKC are performed in two parts: The software part is executed on the processor and handles control tasks such as message communication. The hardware part is implemented on the coprocessor and aims to accelerate modular arithmetic operations.

With high-end processing systems like the Xilinx Zynq-7000 all programmable SoC, it is possible to fully utilize both processing system (PS) and custom coprocessors connected to the PS within a chip. This allow us to exploit both hardware and software design methodologies. In the following, we describe our SoC architectures implemented on Zynq-7000 based embedded SoC platform.

5.1. Zynq-based embedded system

Embedded systems consisting of a processor and a several custom-specific coprocessor are proved to be very efficient for many applications in the field of digital signal processing, cryptography, and image processing. In such an embedded system, custom coprocessor is able to accelerate the computational intensive tasks while the processor is able to perform other tasks. Extensible processing platform with Zynq-7000 family device combines an ARM dual-core Cortex-A9 MPCore processing system with Xilinx 28 nm unified programmable logic architecture (Fig. 3). The ARM dual-core Cortex-A9 processor delivers high-performance capabilities by consuming less power compared to ARM Cortex-A8 solutions. ARM MPCore technology offers a coherent interconnect mechanism within the dual-core CPU platform. The Cortex-A9 processor includes several optional interfaces so that custom peripherals, coprocessors, memory units, and input–output units can be connected to build a complete embedded system on the FPGA via some dedicated tools provided by Xilinx.

Connecting a application specific hardware coprocessor on the processor bus is an elegant way of increasing the efficiency and computational power of an embedded system. Embedded processor based systems within FPGAs exploit the benefits of both software and hardware: flexibility of the software and processing power of the custom hardware. In the design, we transform the slowest part of the system which is computationally intensive modular arithmetic operations into the hardware domain to accelerate the system. The addition of a custom hardware to a processor will consume FPGA resources; however, this often will not be a problem for systems since surplus resources are available in FPGAs. Hence, the careful use of these spare resources provides a powerful, simple, and most importantly cost-free solution for FPGA based systems.

Data transfer overhead has a crucial impact on the execution time even when low latency Zynq buses, AXI based interconnect, are used. Nevertheless, still very high speed-ups are achieved in the proposed work.

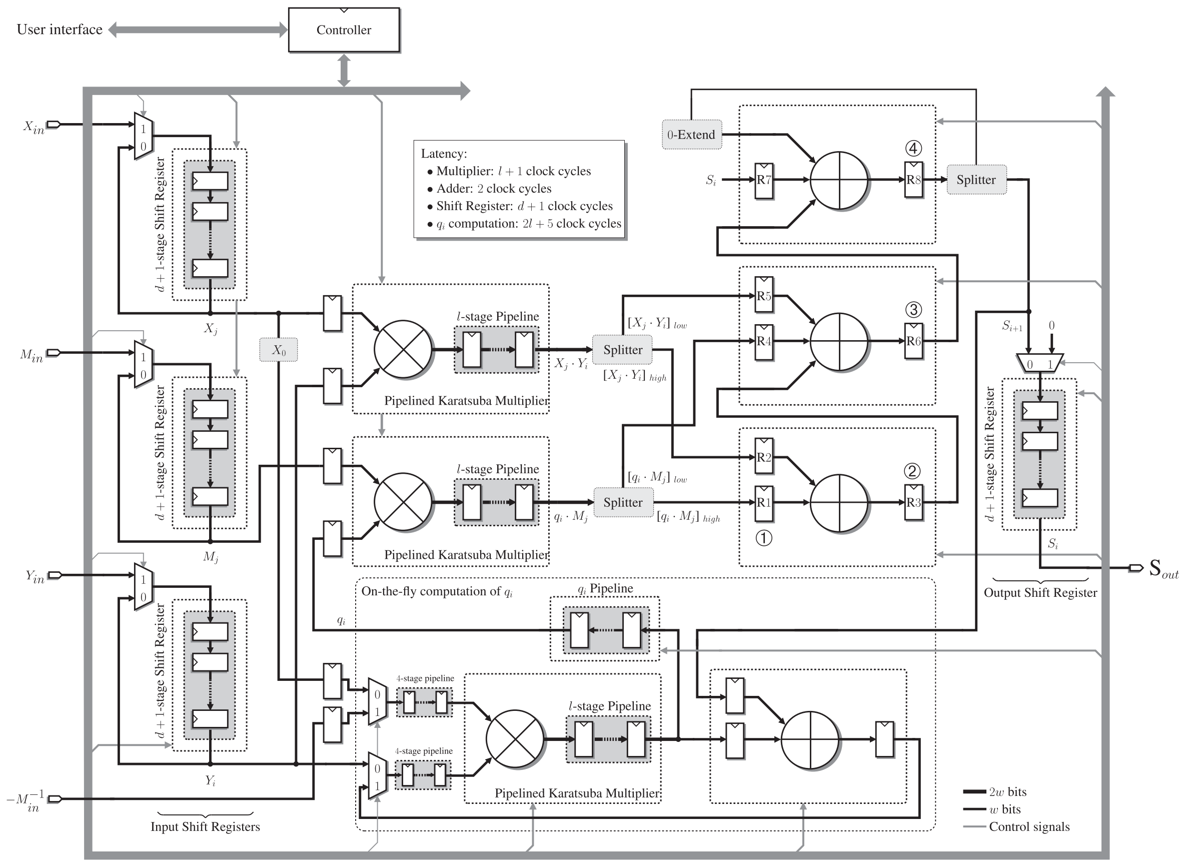


Fig. 1. High-speed Modular arithmetic coprocessor.

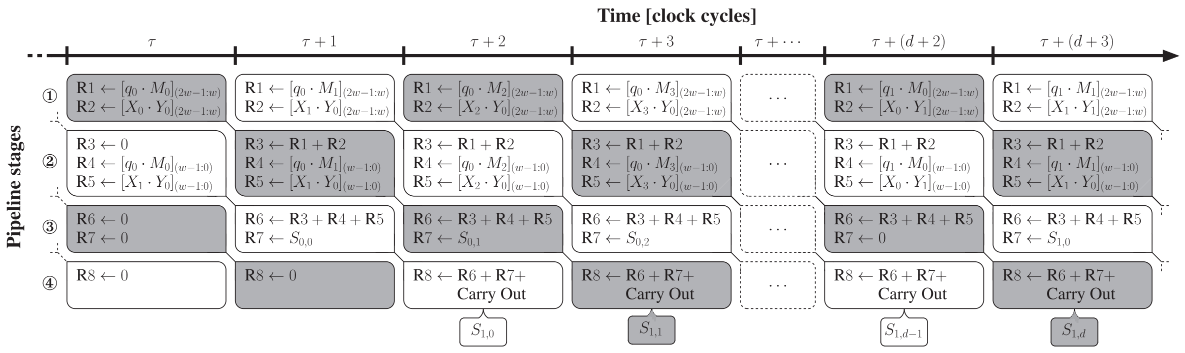


Fig. 2. A view for pipelined computation of high-radix modular multiplication operation.

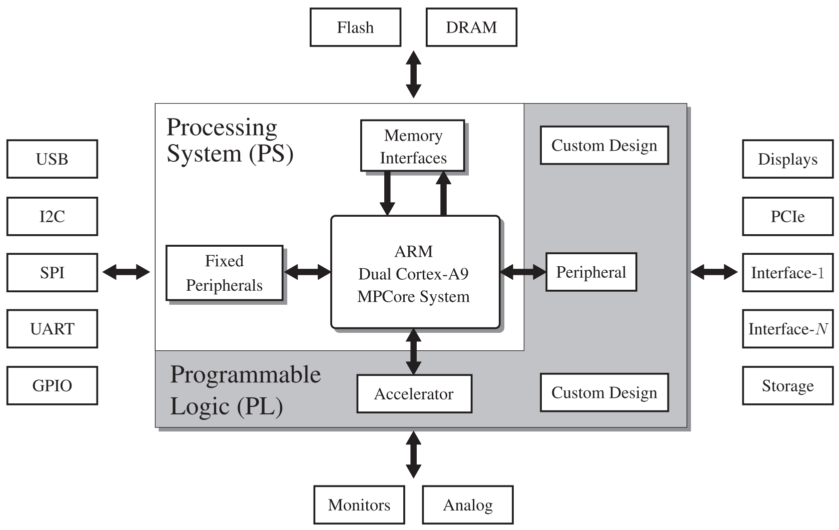


Fig. 3. A generic Zynq-7000 all programmable SoC chip based embedded system and available busses (adapted from Xilinx manuals).

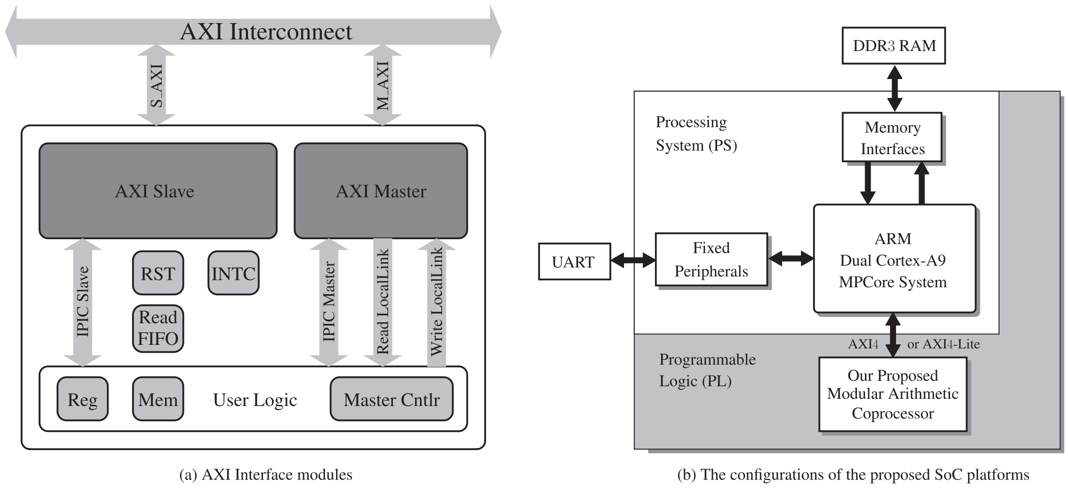


Fig. 4. Communication details of the SoC architectures (adapted from Xilinx manuals).

5.2. Communication methods

On-chip peripherals are connected to the processor via the data and address buses. In Zynq-based extensible processing platform, ARM AXI4 protocol is used for communication between the accelerator and the processor.

There are two types of AXI4 interfaces which we used in this study is as follows:

- AXI4-Lite provides simple, low-throughput memory-mapped communication (for example, to and from control and status registers).

- AXI4 meets the high-performance memory-mapped requirements.

Fig. 4a shows the available components provided by the AXI interconnect for various control mechanisms. We utilize RST, Read FIFO and Reg modules for fast communication between the ARM processor and our accelerator. Fig. 4b shows the connections between the proposed coprocessor and the ARM dual Cortex-A9 processor within Zynq-7000 device via the two types of AXI4 communication protocol.

AXI4-Lite is an area-efficient subset of the AXI protocol. This type of AXI interconnect protocol is able to send only one data word per transaction with less amount of hardware cost. Thus, there is a communication overhead while sending for each data word from the processor to our modular arithmetic accelerator due to the transmitting the control data for setting up each transaction. Simple status and control registers are enough for communication.

AXI4 is a conventional single-address burst interconnect which supports up to 256 data beats per burst, the width of which is system dependent. In this type of AXI communication, the communication overhead is decreased with extra area cost. AXI4 is more suitable for compared to AXI4-Lite since it is capable of carrying more data for each transaction.

6. Results and perspectives

We present our key results that we achieved in this Section. We conduct several experiments to measure the performance of our study. In the following, we first explain our methodology that we follow for performance evaluation of our proposed design for modular arithmetic coprocessor. We then define the metrics used to evaluate the performance and give our empirical results and comparisons with other published studies.

6.1. Methodology

In this study, proposed design strategies and implementations of our modular arithmetic coprocessor target Xilinx FPGAs. Hence, the results of this study are also valuable where FPGA technology comes into play. We mainly use Virtex-7 family of FPGAs to prototype our architecture. The Xilinx 7-family of FPGAs provide architectural elements designed for maximum performance, higher integration, and lower power consumption making a good choice for the high-speed architectures. We select Virtex-7 member of 7-family of FPGAs since it consists of more programmable logic cells together with higher performance compared to the other members, Artix7 and Kintex-7 FPGAs. We captured our modular arithmetic coprocessors in the VHDL language and evaluate the performances of a fully autonomous implementations of our architecture on Xilinx Virtex-7 (7VX330T-3) FPGA. We also use ZedBoard based on the Xilinx's latest Zynq-7000 All Programmable SoC in order to get resource utilization of our SoC architectures. We measure the processing time in ms for SoC architectures with a timer module in software. We synthesized our hardware architecture with Xilinx PlanAhead and Vivado development tools. We also simulate our design with Xilinx ISim Simulator to verify the correctness of the proposed architectures.

6.2. Evaluation metrics

In this study, we used three performance metrics in order to evaluate the performance of the proposed hardware design compared to the others in the literature. They can be explained, as follows:

Area: Required hardware area is very important in efficient hardware designs, especially for the applications where area matters most. We measure the size of the architecture implemented in FPGA in terms of slices and embedded building blocks such as DSP48E1 and Block RAMs.

Frequency: The frequency indicates the operating clock frequency of the proposed hardware. Higher frequency values allow to achieve more performance from the hardware design.

Processing Time: Processing time measures the required time (we consider the worse-case scenario) to complete the whole operation in terms of seconds. It is calculated for performing the operation for one-block of data.

6.3. Results

The evaluation metrics of our modular multiplication and exponentiation coprocessors on FPGA are given in Tables 1 and 2 at different operand sizes. We consider here the less favorable case for modular exponentiation, a high number of modular multiplication operations are performed. In other words, we take the base, exponent, and modulus which all have the same bit length for the modular exponentiation operation. This is significantly important in the comparison of the architectures in terms of latency. Some of the studies in the literature use exponents which have average number of bits for calculating the latency. Therefore, the latency is significantly low. In our study, we select the worse case scenario for latency calculation.

Tables 1 summarizes our synthesis results in terms of the evaluation metrics we defined in Section 6.2. We measured these results with the Xilinx PlanAhead and Vivado tools for the modular multiplication operation. The modular multiplication coprocessor requires for instance 1257 slices and 27 DSP48E1 blocks and achieves processing time of 1.39 μs for the modular multiplication of 1024-bit operands. One can see the scalability of our coprocessor by looking at different modulo sizes where the the same radix size is used. For instance, when radix size is equal to 16, the coprocessors for 512, 1024 and 2048-bit modulo sizes uses same number of 7 DSP48E1 blocks with a small increase in area utilization due to the need of varying sizes of FIFO elements for input and output. Control mechanism does not effect dramatically when the modulo size is increased thanks to the usage of simple counter mechanism.

Table 2 summarizes our synthesis results measured with the Xilinx PlanAhead and Vivado tools for the modular exponentiation operation. The modular exponentiation coprocessor requires for instance 3046 slices and 54 DSP48E1 blocks and achieves competitive processing time of 1.52 ms for the modular exponentiation of 1024-bit operands. The maximum clock frequency of the DSP48E1 is 741 MHz where all registers in the DSP48E1 are used. Thanks to this high-frequency value, we obtained very high-operating frequencies above 400 MHz for the proposed coprocessors. However, for instance, the coprocessor, where modulo size is 512 and radix size is 32, operates on 288 MHz which is relatively small. This is caused by the need to run the pipelined Karatsuba multiplier with a small latency. When the ratio of modulo size to radix size is equal to 32 or above, we have flexibility to run the Karatsuba coprocessor with higher latency which allows us to achieve higher frequencies. We employ some of the embedded registers in the DSP48E1 according to the constraints in our architecture due the our pipeline mechanism (Fig. 1). Due to the this fact and also the controller mechanism delays, we achieved the operating frequencies given in Table 1 and 2. It is possible to improve the achieved frequencies by using more embedded registers but this causes to have a pipeline stall.

In Xilinx's 7VX330T-3 FPGA, there are 1120 DSP48E1 blocks which is lower than other members of Virtex-7 FPGAs. Even if the selected device has the lowest number of DSP48E1 blocks compared the other members of Virtex-7 FPGAs, one can fit 36 and 18 instances for the modular multiplication and exponentiation coprocessors in the FPGA device, respectively. If the embedded multiplier blocks are not employed in the design, surrounding slices of these multipliers can be used for routing purposes or other logic functionalities. Multiplier blocks are surrounded by our coprocessors and the efficiency of resource utilization on FPGA is increased. Thanks to the use of embedded multipliers in pipelined mode, significant performance improvement in terms of latency and frequency is attained. This leads to substantial increase at the attained performance for parallel modular arithmetic operations which are extensively required in many secure protocols such as RSA with homomorphic property.

Table 3 illustrates the results of latest hardware architectures for the modular exponentiation operation (see for instance [25] for various hardware architectures). We consider the least favorable case for modular exponentiation operation where base, exponent and modulus have all same size. Some of the reported studies in Table 3 take into account that the average number of modular multiplications is required for the exponentiation. This work attains very good performance results compared to the other studies in terms of processing time which is the main optimization goal in this study. We achieve for instance 1.52 ms for 1024 bit operands whereas all other reported studies are less than in processing time of one-block of data. Our proposed architecture is much better than some of the proposed designs with the cost of extra usage of DSP48E1 blocks. Fortunately, today's FPGA devices have a large amount of programmable logic components and embedded DSP blocks as it can be seen in our selected FPGA device.

The latency of our Montgomery modular multiplication operation is calculated using the following equation.

$$\text{Latency} = (d+1) * (d+1) + (l+1)$$

When radix size is increased, the depth of the multiplier is decreased and the latency is decreased significantly. Thanks to this, we are able to operate on higher radix sizes and having much less latency compared to the others.

In order to show the effects of our hardware architectures in a real embedded system, we build a SoC design whose framework is explained in detail in Section 5. We compare the empirical findings obtained from all different SoC designs that we built. We prototype our Zynq-7000 based SoC designs using Xilinx PlanAhead tool.

Pure software SoC design is in fact consists of only dual core Cortex-A9 processor and some I/O hardware modules without any accelerator core for modular arithmetic. Modular arithmetic operations are realized by means of pure software. The other two SoC designs include our modular arithmetic coprocessor with different communication mechanisms. One of them is connected to Cortex-A9 processor with AXI4-Lite and the other one is connected with AXI4 interface. We give the performance results of our all different SoC designs in Table 4. As seen from the table, our speed-up is for instance about 121 for 1024-bit operands. The speed-up achieved with our modular arithmetic coprocessor can be increased where parallel execution of modular arithmetic operations is possible. Such a scenario always exists in many public-key cryptosystems since they operate on high-dimensional matrices of data with privacy. Such evaluation is given before in this Section.

According to the results of our proposed modular arithmetic coprocessor, it can be seen that significant performance improvement is attained. Furthermore, we also evaluate our modular arithmetic coprocessor on different SoC platforms described in Section 5 to emphasize on the contribution of our proposed hardware architecture from systems perspective.

This study contains three important features.

- The device specific units in FPGA are well-suited for the pipelined ALUs for our overall design.
- We use very high radix Montgomery algorithm together with compact large pipelined Karatsuba multipliers.
- We also keep the pipeline always full to increase the processing time thanks to the our scheduling.

7. Conclusion

We take advantage of both high-radix Montgomery modular multiplication and Karatsuba algorithms and harness the intrinsic parallelism of these algorithms to design a pipelined ALU. We also interleave independent tasks in the algorithms in order to achieve a very tight scheduling. The design philosophy we proposed in this study led us to develop a high-performance scalable coprocessor for modular arithmetic operations including modular multiplication and exponentiation at different operand sizes. From our point of view, the main advantage of this method over other existing methods is that the algorithm combines the efficiency of both Montgomery and Karatsuba algorithms in a compact manner using built-in device specific hardware modules placed in FPGAs.

Despite the various control signals required for the different steps of pipelined computation of Algorithm 4, our control unit remains compact: almost all control signals are generated by means of a counter, a start signal and its delayed versions. Thanks to an alternative description of 2^w radix Montgomery modular multiplication and a careful organization of the pipelined datapath, we manage to implement the modular arithmetic operations (multiplication and exponentiation) without any pipeline stall. The key element of our approach to achieve high-throughput design is to take advantage of the parallelism of modular arithmetic operations to.

- deeply pipeline the ALU to achieve a high clock frequency;
- find parallelism between independent tasks of the modular arithmetic operations.
- decrease the data dependencies and hazards to achieve more parallelism and exploit this parallelism in hardware better.
- reuse same resources for different executions of the algorithm.
- design pipelined datapath for the independent tasks so that each pipeline units can operate on different set of inputs at the same time.

This study also reveals a better understanding of the modular arithmetic operations in terms of computation for FPGA based applications and also includes empirical performance analysis of the algorithm from hardware perspective. Accelerating the computation of cryptographic algorithms especially for public key cryptography to protect network traffic and confidential data within some area constraints is rapidly becoming significant in today's processor technology.

Acknowledgements

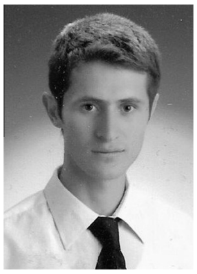
The authors would like to thank the anonymous reviewers for their valuable comments. They would also like to thank Xilinx and the Xilinx University Program for its generous donation of materials in terms of design tools.

References

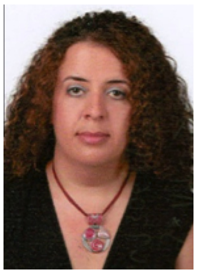
[1] I. San, N. At, On increasing the computational efficiency of long integer multiplication on FPGA, in: Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, Washington, DC, USA, 2012, pp. 1149–1154.

[2] T. Güneysu, Utilizing hard cores of modern FPGA devices for high-performance cryptography, Journal of Cryptographic Engineering 1 (1) (2011) 37–55.

[3] F. de Dinechin, B. Pasca, Large multipliers with fewer DSP blocks, in: Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, 2009, pp. 250–255.
[4] J. von zur Gathen, J. Shokrollahi, Efficient FPGA-based karatsuba multipliers for polynomials over f2, in: B. Preneel, S.E. Tavares (Eds.), Selected Areas in Cryptography, 2005, pp. 359–369.
[5] E.-H. Wajih, M. Mohsen, Z. Medien, B. Belgacem, Efficient hardware architecture of recursive Karatsuba-Ofman multiplier, in: Design and Technology of Integrated Systems in Nanoscale Era, 2008. DTIS 2008. 3rd International Conference on, 2008, pp. 1–6.
[6] S. Bo, K. Kawakami, K. Nakano, Y. Ito, An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the FPGA, International Journal of Networking and Computing 1 (2) (2011) 277–289.
[7] T. Blum, C. Paar, High-radix Montgomery modular exponentiation on reconfigurable hardware, IEEE Transactions on Computers 50 (7) (2001) 759–764.
[8] P.L. Montgomery, Modular multiplication without trial division, Mathematics of Computation 44 (1985) 519. 519.
[9] A.A. Karatsuba, Y. Ofman, Multiplication of multidigit numbers by automata 145 (1962) 293–294 (english translation in Soviet Physics Doklady 7 (1963) 595–596).
[10] A. Karatsuba, Y. Ofman, Multiplication of many-digital numbers by automatic computers, Proceedings of the USSR Academy of Sciences 145 (1962) 293–294.
[11] H. Orup, Simplifying quotient determination in high-radix modular multiplication, in: Computer Arithmetic, 1995, Proceedings of the 12th Symposium on, 1995, pp. 193–199.
[12] C.D. Walter, Space/time trade-offs for higher radix modular multiplication using repeated addition, IEEE Transactions on Computers 46 (2) (1997) 139–141.
[13] A.F. Tenca, Ç. K. Koç, A Scalable Architecture for Montgomery Multiplication, in: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems, CHES '99, Springer–Verlag, London, UK, UK, 1999, pp. 94–108.
[14] A.F. Tenca, G. Todorov, c. K. Koç, High-radix design of a scalable modular multiplier, in: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems, CHES '01, 2001, pp. 185–201.
[15] J.-L. Beuchat, J.-M. Muller, Automatic Generation of modular multipliers for FPGA applications, IEEE Transactions on Computers 57 (12) (2008) 1600–1613.
[16] A. Weimerskirch, C. Paar, Generalizations of the Karatsuba algorithm for efficient implementations, IACR Cryptology ePrint Archive (2006) 224.
[17] D.J. Bernstein, Multidigit multiplication for mathematicians, 2001. URLhttp://www.scribd.com/doc/3669457/Multidigit-Multiplica tion-for-mathematicians
[18] N. Nedjah, L. de Macedo Mourelle, A review of modular multiplication methods and respective hardware implementation, Informatica (Slovenia) 30 (1) (2006) 111–129.
[19] D. Suzuki, T. Matsumoto, How to maximize the potential of FPGA-based DSPs for modular exponentiation, IEICE, Transactions 94-A (1) (2011) 211–222.
[20] S. Gao, D. Al-Khalili, N. Chabini, Efficient scheme for implementing large size signed multipliers using multigranular embedded DSP blocks in FPGAs, International Journal of Reconfigurable Computing 1 (2009). 1:1–1:11.
[21] G.C.T. Chow, K. Eguro, W. Luk, P. Leong, A Karatsuba-based Montgomery multiplier, in: Proceedings of the 2010 International Conference on Field Programmable Logic and Applications, FPL '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 434–437. doi:10.1109/FPL.2010.89. URL http://dx.doi.org/10.1109/FPL.2010.89
[22] B. Song, Y. Ito, K. Nakano, CRT-based DSP decryption using Montgomery modular multiplication on the FPGA, in: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, 2011, pp. 532–541.
[23] J. Bautista, O. Alvarado-Nava, F. Perez, A mathematical co-processor of modular arithmetic based on a FPGA, in: Technologies Applied to Electronics Teaching (TAEE), 2012, 2012, pp. 32–37.
[24] N. At, J.-L. Beuchat, E. Okamoto, I. San, T. Yamazaki, A low-area unified hardware architecture for the AES and the cryptographic hash function Grøstl, Cryptology ePrint Archive, Report 2012/535, 2012.
[25] L. Batina, S.B. Örs, B. Preneel, J. Vandewalle, Hardware architectures for public key cryptography, Integrative VLSI Journal 34 (1–2) (2003) 1–64.
[26] K. Nakano, K. Kawakami, K. Shigemoto, RSA encryption and decryption using the redundant number system on the FPGA, in: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, Washington, DC, USA, 2009, pp. 1–8.
[27] S. Tang, K. Tsui, P. Leong, Modular exponentiation using parallel multipliers, in: Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on, 2003, pp. 52–59.



Mr. San received his B.Sc. degree from Electrical and Electronics Engineering Department at Anadolu University, Turkey, in 2008. He is currently doing his PhD in the same department. His main interests are architecture design for communication systems with special focus on efficient hardware implementations for cryptographic applications.



Dr. At received her B.Sc. degree from Electrical and Electronics Engineering Department at Anadolu University; M.Sc. degree from Electrical and Computer Engineering Department at the University of California, Davis (1998); M.Sc. degree from the Georgia Institute of Technology (2001); and Ph.D. degree from Electrical and Electronics Engineering Department at Anadolu University (2005). She held a visiting researcher position at the Selmer Center, University of Bergen. She is currently a faculty member in Electrical and Electronics Engineering Department at Anadolu University. Her research interests include communication systems, information security, and signal processing.