# Architecture Optimizations for the RSA Public Key Cryptosystem: A Tutorial

Aaron E. Cohen and
Keshab K. Parhi

**Abstract**

The Rivest Shamir Adleman (RSA) cryptosystem, named after its creators, is one of the most popular public key cryptosystems. The RSA cryptosystem has been utilized for e-commerce, various forms of authentication, and virtual private networks. The importance of high security and faster implementations paved the way for RSA crypto-accelerators, hardware implementations of the RSA algorithm. This work consists of describing various approaches to implementing RSA crypto-accelerators based on the "textbook" version of the RSA cryptosystem and comparing their area requirements. Many of the techniques described here have applications elsewhere such as in digital signal processing and error correcting codes. This paper presents the four fundamental architectures: the bit-serial squaring architecture, two bit-serial systolic array modular multiplication architectures, and the interleaved modular multiplication architecture.

## I. Introduction

In the mid 1970's early public key cryptosystems were developed to provide a solution for two parties to communicate in a secure manner over an insecure channel [1], [2]. The application of public key cryptosystems to communication increased almost exponentially with the invention of the Internet. A few examples are the numerous financial transactions that occur over the Internet every day, remote telework through virtual private networks, and Voice over Internet protocol (VoIP). Public key cryptosystems are used to establish secure communication and to provide non-repudiation

*Digital Object Identifier 10.1109/MCAS.2011.942747*
*Date of publication: 22 November 2011*

through the use of digital signatures. Many public key cryptosystems base their security on the difficulty of solving a known hard problem such as discrete logarithms or integer factoring. Thus they are assumed to be very secure; however, many researchers have shown that breaking cryptosystems is not necessarily as difficult as solving these known hard problems. Recently, several side channel attacks have been investigated which substantially reduce the mean time to disclosure of the secret key [3], [4].

Although computation power has increased with Moore's law, the large increase in computation costs associated public key cryptosystems has put a significant strain on available computing resources. Thus, there is a growing need for hardware acceleration of public key cryptosystems to reduce the burden of using them. Public key cryptosystems have become ubiquitous in computing devices. Not only have servers but also embedded systems have forced designers to add additional hardware for cryptosystems referred as *crypto-accelerators*. Crypto-accelerators are very promising as they typically achieve better performance and better power efficiency than a software implementation on a generic processor. A crypto-accelerator can be implemented in a field programmable gate array (FPGA), a digital signal processor (DSP) or an application specific integrated circuit (ASIC).

Many of the techniques for optimizing the speed of crypto-accelerators were originally discovered for DSP applications [5]. However, many other techniques have been well known to mathematicians and logic designers since the early 80's. In this paper, the fundamental circuit design techniques for developing crypto-accelerators for public key cryptography will be briefly covered, followed by diagrams or illustrations which will aid in understanding these important concepts. The intended audience is anyone involved in logic design or who has a basic understanding of logic design and who has an interest in hardware implementations of public key cryptosystems such as the RSA or Diffie-Hellman cryptosystems [1], [2].

In [6], a high speed RSA implementation is proposed that makes use of both Booth's technique and Montgomery's Multiplication technique. High radix Montgomery multiplication is studied in [7]. This work shows that high radix Montgomery multiplication is able to achieve significantly faster public key cryptography speeds with considerable increase in gate area. Another high speed approach is presented in [8]. This work makes use of carry save adders to achieve higher speed and parallelism. Three different implementation techniques were presented in [9]. They are sequential, parallel, and systolic.

Integrating RSA into general purpose processors has been a challenge since Barrett first proposed an RSA implementation for general DSP processors [10]. More recent work has shown how to integrate RSA into MIPS processors [11].

Prior work on side channel attack resistant RSA has been published in [12]–[17]. These works have examined the difficulties with protecting the Chinese Remainder Theorem (CRT) operation and the Montgomery multiplication operation from side channel attacks. These include both fault attacks and passive power analysis monitoring attacks.

This paper is organized as follows. Section II provides a description of serial squaring. Section III provides a description of systolic array implementation of Montgomery multiplication. Section IV provides a description of the interleave architecture design. Section V provides a comparison of the different architectures. Finally, Section VI provides a summary and conclusion.

## II. Serial Squaring

This section reviews the serial squaring architecture concept discussed in [18]. The general purpose serial squaring architecture for integers will be presented because it can be combined with a separate modular reduction unit to obtain the desired result for squaring Montgomery domain numbers. Alternatively, the reduction unit can be integrated with serial squaring as suggested in [18].

Some RSA crypto-acceleration architectures require one operand in the modular multiplication to be input in serial and the other in parallel. The results are generated serially. A problem arises when using successive squaring operations with either the well known H-Algorithm or the L-Algorithm [1], [19], [20] because both require an architecture that can take both inputs serially and produce a result serially due to data dependencies. Serial squaring is a method that allows us to perform the squaring operation with both inputs fed serially. This is best illustrated graphically as shown in Fig. 1.

The most obvious approach to implementing serial squaring is to simply shift everything in the lower triangle of the multiplication by one position followed by subtracting the bits which should not have been shifted. The bits that should not have been added twice form the value

$$S = a_3, 0, a_2, 0, a_1, 0, a_0.$$

*Aaron E. Cohen and Keshab K. Parhi are with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455 USA (email: {cohen082, parhi}@umn.edu).*

**Not only have servers but also embedded systems have forced designers to add additional hardware for cryptosystems referred as *crypto-accelerators*.**

Then serial squaring reduces to computing the lower triangular (LT) numbers weighted by two minus $S$. This requires one serial input of $a_{n-1}, 0, \ldots, a_1, 0, a_0$ delaying the input corresponding to the distance between the two values that connect to the AND logic gates. The following equation summarizes the basic computation performed by the serial squarer

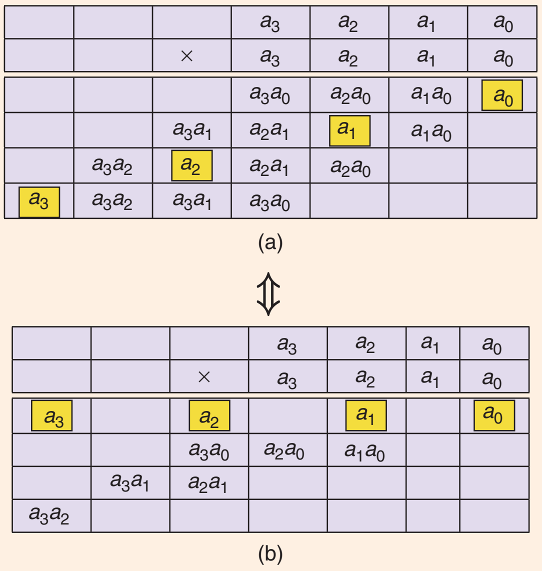
$$\text{Serial Squaring} = 2 \times LT - S = 2 \times LT + S' + 1,$$

where $LT$ represents the lower triangular numbers, and $S'$ represents the one's complement of $S$ such that $S' + 1$ is the two's complement of $S$.

The serial squarer technique requires an additional input on the first adder to handle the serial subtraction and one additional cycle due to the times two operation on the serial input. To ensure the carry values are cleared and the one is added for the subtraction, a carrykill control signal is used. The first adder uses an OR logic gate with the inverted control signal such that its carryin value is equal to (delayed_carryout or carrykillbar). The other adders use the carryin value equal to (delayed_carryout and carrykill). Fig. 2 illustrates this design.

One thing to note is that the addition terms are not added until the first $a_0$ term reaches the last adder. This is equivalent to a delay of (size-1)*2 cycles. However, the subtraction term is added in one cycle before the $a_0$ term reaches the last adder. This is equivalent to a delay of (size-1)*2-1 cycles.

The critical path of the serial squarer architecture is illustrated in red in Fig. 2. This critical path is equal to $t_{CP} = t_{INV} + t_{OR} + t_{FA}$, where $t_x$ represent time for element $x$, CP stands for the critical path, INV stands for an invertor



| | | | | | | |
|---|---|---|---|---|---|---|
| | | | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| | | $\times$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| | | | $a_3a_0$ | $a_2a_0$ | $a_1a_0$ | $a_0$ |
| | | $a_3a_1$ | $a_2a_1$ | $a_1$ | $a_1a_0$ | |
| | $a_3a_2$ | $a_2$ | $a_2a_1$ | $a_2a_0$ | | |
| $a_3$ | $a_3a_2$ | $a_3a_1$ | $a_3a_0$ | | | |

(a)

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| | | $\times$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| $a_3$ | | $a_2$ | | $a_1$ | | $a_0$ |
| | | $a_3a_0$ | $a_2a_0$ | $a_1a_0$ | | |
| | $a_3a_1$ | $a_2a_1$ | | | | |
| $a_3a_2$ | | | | | | |

(b)

**Figure 1.** Illustration of weighting lower triangular terms for serial squaring. (a) Standard squaring of 4 bit numbers. (b) Squaring after weighting the lower triangular numbers.

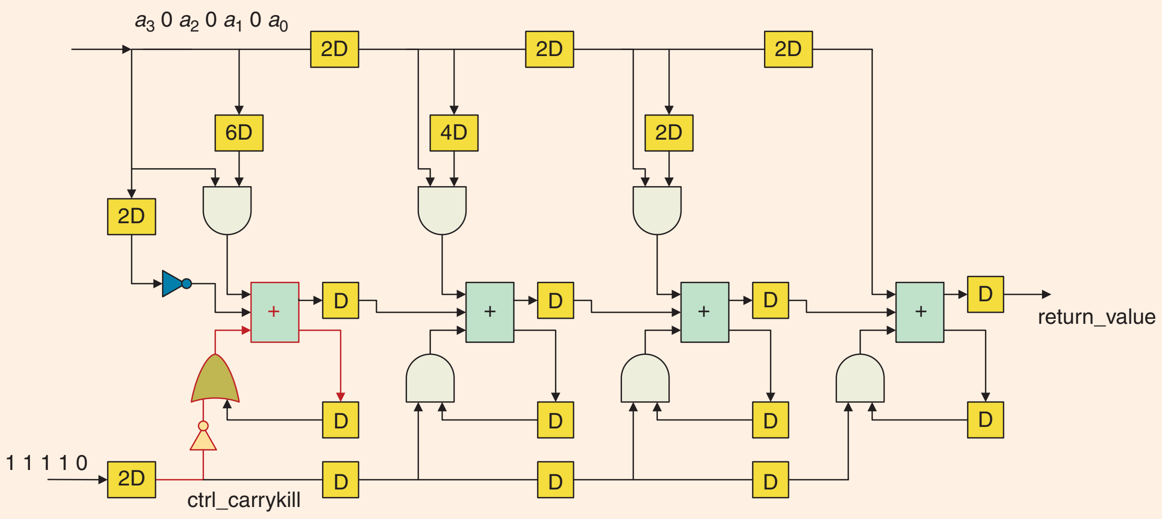


**Figure 2.** Squaring with serial squarer architecture for 4 bit word length.

**The vector merging unit converts the carry save (redundant) format answer into a 2's complement result.**

logic gate, OR stands for an OR-logic gate, and FA stands for a full adder implementation.

Table 1 shows the synthesis results for implementing the serial squarer design using 4-bit, 512-bit, and 1,024-bit operands. These results were synthesized using Xilinx ISE Webpack version 12.1 for the Xilinx Virtex xc4vlx25-12sf363. The Xilinx software has a built-in limit on the generate command which prevented obtaining results for operands larger than 1,024-bits. The synthesized minimum period for a 4 bit implementation of the serial squarer is 1.433ns with a maximum frequency of 698.032 MHz on the Virtex xc4vlx25-12sf363. The synthesized minimum period for a 1,024-bit implementation of the serial squarer is 2.832ns with a maximum frequency of 353.107 MHz on the Virtex xc4vlx25-12sf363.

This serial squaring architecture was designed for normal integer based squaring and does not make use of Montgomery reduction. If numbers are in the Montgomery domain then either a separate Montgomery reduction unit will be needed after the serial squaring operation or the Montgomery reduction will need to be integrated into the serial squaring architecture. Integrating requires checking the least significant bit (LSB) and adding $N$ when the LSB is one following the standard Montgomery reduction method illustrated in the next section.

A typical implementation of the RSA algorithm's modular exponentiation operation would require $O(\log_2(E))$ squaring operations, where $E$ represents the exponent in the modular exponentiation. The serial squarer uses $n + n - 1$ cycles for the operand with an initial latency of $n + n - 1$ cycles. If the serial squarer and a serial multiplier directly feed a Montgomery reduction unit/accumulator, then the total cycle complexity is approximately $\log_2(E) * (2n-1) + 2n-1$. The total speed per modular exponentation then becomes $(\log_2(E) * (2n-1) + 2n-1) * t_{CP}$. If the critical path is less than 4.766 ns then the RSA modular exponentiation operation will complete in less than 10 ms. This leads to over 100 RSA operations per second.

## III. Montgomery Multiplier Systolic Implementations

The most popular technique for modular multiplication is no doubt Montgomery's $N$-Residue technique [21]. Not only does it lend itself to fast systolic architectures but unlike Barrett's estimation it does not require any precomputation and unlike sign estimation it requires far fewer hybrid adders. Barrett's estimation is well known among computer programmers; however, it is not as well known that it can be implemented in a highly parallel fashion. A majority of papers dealing with Barrett's estimation typically use Barrett's estimation technique for embedded systems rather than high speed parallelized hardware implementations. Finally, residue number systems with Montgomery's $N$-Residues lead to high speed parallelized implementations.

Typical implementations of the REDC() function from Montgomery's paper [21] consist of systolic array architectures [22] and digit serial designs.

The beginning of designing systolic array architectures [5] is to fit the problem (REDC(A,B)) into the typical dataflow graph detailed down to the bit level. Therefore it is necessary to convert the additions, divisions, reductions, etc. into bitwise operations. Folding and pipelining are then used to generate a fast bit-serial architecture.

Analyzing the REDC() function, the (mod 2) operation refers to the LSB of the addition result. The (div 2) operation is a shift right operation. The addition operation can be implemented in a redundant format or a non-redundant format, which determines the direction of the carry bits. The dataflow graph is illustrated in Fig 3.

There has been a lot of interest in designing systolic array modular multipliers. Designing a pipelined bit-serial $N$-Residue multiplier (Montgomery multiplier) can be accomplished by using the following projection vector $(d^T = (0,1))$, scheduling vector $(s^T = (2,1))$, and processor

Table 1.
Squarer Synthesis Results for 4-bits to 1,024-bits.

| Bit | Slice | Slice-FF | 4 input LUT | Bonded IOBs | GCLKS |
|---|---|---|---|---|---|
| Maximum | 10752 | 21504 | 21504 | 240 | 32 |
| 4 | 10 | 19 | 11 | 5 | 1 |
| 512 | 1627 | 3119 | 1568 | 5 | 1 |
| 1,024 | 3258 | 6247 | 3136 | 5 | 1 |

**ALGORITHM: REDC(A, B) FUNCTION**

```
$X_0 = 0$
for ( $i = 0, i < n; i++$ )
$q_i = (X_i + a_i B) \bmod 2$
$X_{i+1} = (X_i + a_i B + q_i N)$ div 2
return($X_n$)
```

**The most popular technique for modular multiplication is no doubt Montgomery's *N*-Residue technique.**

space vector ($p^T = (1,0)$) [5]. Then using this information and applying to the DFG from Fig. 3 will generate the design in Fig. 4. This design is composed of a processing element followed by a vector merging unit. The processing element performs a component of the Montgomery multiplication. The vector merging unit converts the carry save (redundant) format answer into a 2's complement result.

The critical path of the systolic architecture is illustrated in red in Fig. 5. This critical path is equal to $t_{CP} = t_{FA} + t_{XOR} + t_{MUX} + t_{AND} + t_{FA}$, where $t_x$ represent time for element $x$, CP stands for the critical path, XOR stands for an exclusive OR logic gate, MUX stands for a multiplexor implementation, AND stands for an AND logic gate, and FA stands for a full adder implementation.

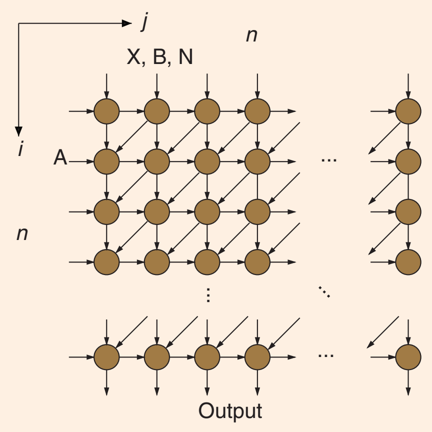
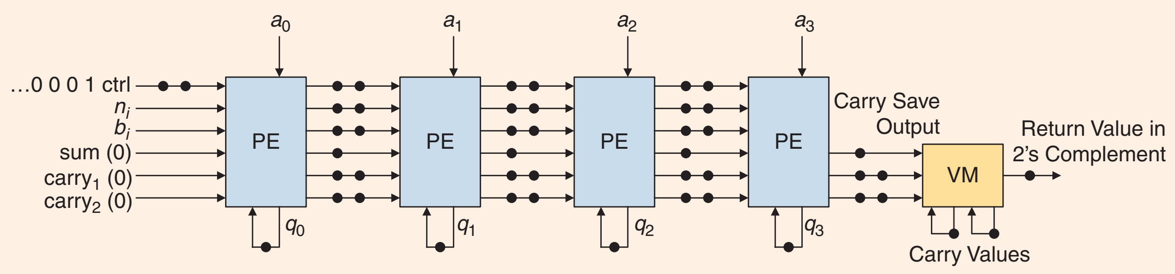


**Figure 3.** Data flow graph (DFG) of the REDC() function.

Table 2 shows the synthesis results for implementing the systolic design using 4-bit, 512-bit, and 1,024-bit operands. These results were synthesized using Xilinx ISE Webpack version 12.1 for the Xilinx Virtex xc4vlx25-12sf363. The Xilinx software has a built-in limit on the generate command which prevented obtaining results for operands larger than 1,024-bits. The synthesized minimum period for a 4 bit implementation of the systolic architecture is 1.936ns with a maximum frequency of 516.556 MHz on the Virtex xc4vlx25-12sf363. The synthesized minimum period for a 1,024 bit implementation of the systolic architecture is 1.906ns with a maximum frequency of 524.728 MHz on the Virtex xc4vlx25-12sf363.

A typical implementation of the RSA algorithm's modular exponentiation operation would require $O(\log_2(E))$ modular squaring operations, where $E$ represents the exponent in the modular exponentiation. The systolic architecture uses $2n + 1$ cycles. If two systolic multipliers are used in parallel, then the total cycle complexity is approximately $\log_2(E) * (2n + 1)$. The total speed per modular exponentiation then becomes $(\log_2(E) * (2n + 1)) * t_{CP}$. If the critical path is less than 4.766 ns then the RSA modular exponentiation operation will complete in less than 10 ms for $n = 1{,}024$. This leads to over 100 RSA operations per second.

Alternatively, if the projection vector is chosen as ($d^T = (1,0)$), scheduling vector ($s^T = (2,1)$), and processor space vector ($p^T = (0,1)$) this generates a different design [5]. Then using this information and applying to the DFG from Fig. 3 will generate the design in



| Signals | $e^T$ | $p^Te$ | $s^Te$ |
|---|---|---|---|
| $X$ (sum) | (1,–1) | 1 | 1 |
| $B$, $N$, $X$ (carry) | (1,0) | 1 | 2 |
| $a_i$, $q_i$ | (0,1) | 0 | 1 |

**Figure 4.** Bit-serial *N*-residue multiplier for 4 bit word length.

Fig. 6. Unlike the previous design, this will lead to a 2-slow processing architecture which performs two time multiplexed Montgomery multiplications together before the results of both become available. From [5], this is calculated as follows: the hardware utilization efficiency (HUE) equals $1/|s^T d| = 1/2$ or 50% for this example.

The critical path of the alternative systolic architecture is illustrated in red in Fig. 7. This critical path is equal to $t_{CP} = t_{INV} + t_{XOR} + t_{FA} + t_{FA}$, where $t_x$ represent time for element $x$, CP stands for the critical path, XOR stands for an exclusive OR logic gate, INV stands for an inverter logic gate, and FA stands for a full adder implementation.

Table 3 shows the synthesis results for implementing the systolic design using 4-bit, 512-bit, and 1,024-bit operands. These results were synthesized using Xilinx ISE Webpack version 12.1 for the Xilinx Virtex xc4vlx25-12sf363. The Xilinx software has a built-in limit on the generate command which prevented obtaining results for operands larger than 1,024-bits. The synthesized minimum period for a 4 bit implementation of the systolic architecture is 1.536 ns with a maximum frequency of 650.978 MHz on the Virtex xc4vlx25-12sf363. The synthesized minimum period for a 1,024 bit implementation of the systolic architecture is 1.536 ns with a maximum frequency of 650.978 MHz on the Virtex xc4vlx25-12sf363.

A typical implementation of the RSA algorithm's modular exponentiation operation would require $O(\log_2(E))$ modular squaring operations, where $E$ represents the exponent in the modular exponentiation.

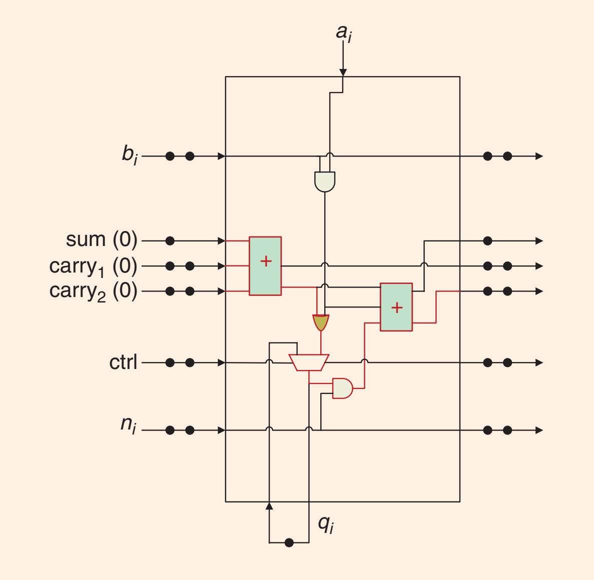
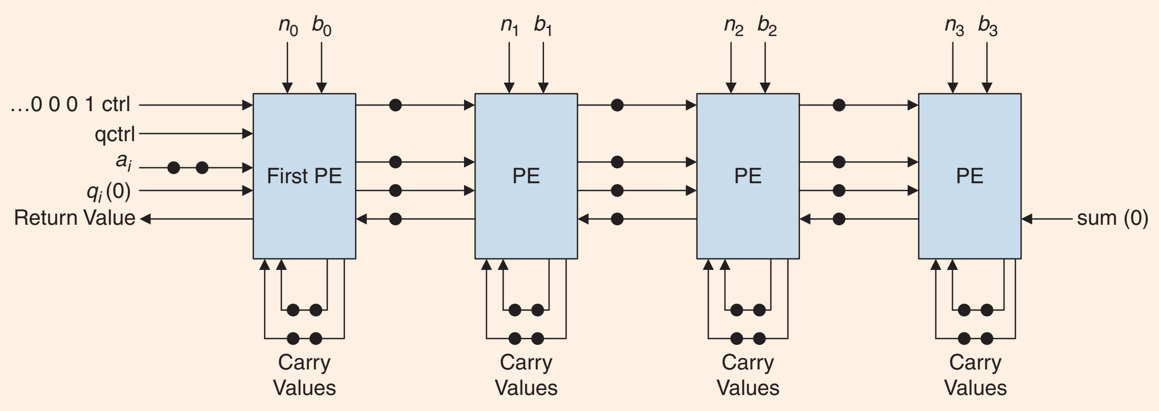


**Figure 5.** Individual processing element (PE) for the systolic array from Fig. 4.

**Table 2.**
**Systolic synthesis results for 4-bits to 1,024-bits.**

| Bit | Slice | Slice-FF | 4 Input LUT | Bonded IOBs | GCLKS |
|---|---|---|---|---|---|
| Maximum | 10752 | 21504 | 21504 | 240 | 32 |
| 4 | 26 | 45 | 31 | 10 | 1 |
| 512 | 4689 | 6141 | 6141 | 518 | 1 |
| 1,024 | 9384 | 12285 | 12285 | 1030 | 1 |



| Signals | $e^T$ | $p^T e$ | $s^T e$ |
|---|---|---|---|
| $X$ (sum) | (1,–1) | –1 | 1 |
| $B$, $N$, $X$ (carry) | (1,0) | 0 | 2 |
| $a_i$, $q_i$ | (0,1) | 1 | 1 |

**Figure 6.** Alternative bit-serial $N$-residue multiplier for 4 bit word length.

**The main difference in performance between these two designs is that the first design has a hardware utilization efficiency (HUE) of 1 whereas the second design has a HUE of 1/2.**

The systolic architecture uses $2n+1$ cycles. If two systolic multipliers are used in parallel, then the total cycle complexity is approximately $\log_2(E)*(2n+1)$. The total speed per modular exponentation then becomes $(\log_2(E)*(2n+1))*t_{CP}$. If the critical path is less than 4.766 ns then the RSA modular exponentiation operation will complete in less than 10 ms for $n = 1{,}024$. This leads to over 100 RSA operations per second.

The main difference in performance between these two designs is that the first design has a hardware utilization efficiency (HUE) of 1 whereas the second design has a HUE of 1/2. If multiple modular multiplication operations can be performed in parallel, then designers can choose between either of these designs. If modular multiplications must be performed sequentially due to dependencies then the first design is preferred. This later case can occur for elliptic curve point arithmetic operations.

## IV. Interleaving Montgomery Multiplier [18]

To improve performance, the intermediate result of the modular multiplication is typically left in carry-save format. Unfortunately this modification requires a 4-to-2 adder in the loop bound of the REDC(A, B) function. By rewriting the equation to change the dependencies like the method of look-ahead [5] the loop bound can be reduced as follows.

$$2X_{i+1} = X_i + a_iB + q_iN$$
$$2X_{i+2} = X_{i+1} + a_{i+1}B + q_{i+1}N$$
$$4X_{i+2} = 2X_{i+1} + 2a_{i+1}B + 2q_{i+1}N$$
$$\begin{aligned}4X_{i+2} &= X_i + a_iB + q_iN + 2a_{i+1}B + 2q_{i+1}N\\ &= X_i + (2a_{i+1} + a_i)B + (2q_{i+1} + q_i)N.\end{aligned}$$

Next is the new modified interleaved REDC() algorithm.

Table 3.
Alternative systolic synthesis results for 4-bits to 1,024-bits.

| Bit | Slice | Slice-FF | 4 Input LUT | Bonded IOBs | GCLKS |
|---|---|---|---|---|---|
| Maximum | 10,752 | 21,504 | 21,504 | 240 | 32 |
| 4 | 16 | 30 | 22 | 14 | 1 |
| 512 | 2,180 | 4,094 | 2,562 | 1,030 | 1 |
| 1,024 | 4,357 | 8,190 | 5,122 | 2,054 | 1 |

**ALGORITHM: INTERLEAVED REDC(A, B) FUNCTION**

```
$X_{-1} = X_0 = 0$
$a_{-1} = q_{-1} = 0$
for ($i = -1; i <= n-2; i++$)
    $q_{i+1} = (X_{i+1} + a_{i+1}B)$ mod 2
    $X_{i+2} = (X_i + (2a_{i+1} + a_i)B + (2q_{i+1} + q_i)N)$ div 4
return ($X_n$)
```

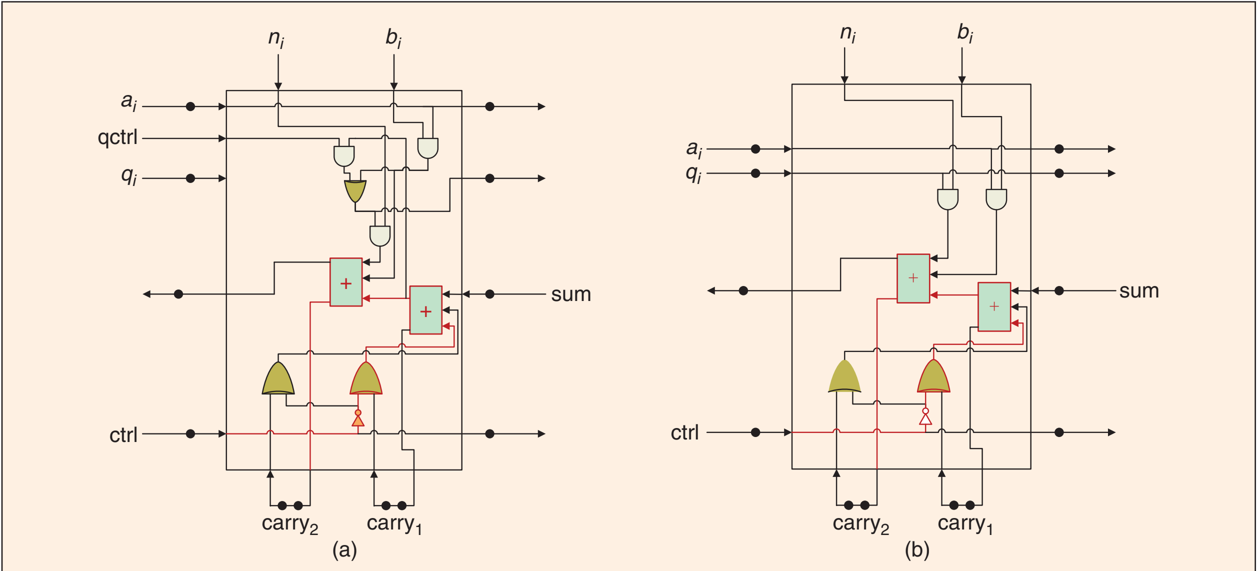


**Figure 7.** Processing elements: (a) first processing element (b) regular processing element.

**The primary disadvantage with implementing the interleaved architecture is that adding negative values by inverting a number and adding 1 to the carry input requires that the carry input value after dividing by 4 should be zero.**

In order to implement $(2a_{i+1}+a_i)X$ efficiently Booth encoding is used [5]. Normally pre-computing $(0*X, 1*X, 2*X, 3*X)$ is required but unfortunately computing times three $(3*X)$ requires a shift plus an addition. This method requires the carry to ripple through all $n$ bits. However, the interleaved method uses the multiples $(0*X, 1*X, 2*X, -1*X)$ that are easy to compute through the use of shifting, anding, and complementing. By using the interleaved method, the addition of $3*X$ is not needed. In essence, this is a type of recoding method, namely Booth recoding. Table 4 illustrates this concept. First lets define the variables used in the recoding table.

| | |
|---|---|
| $X =$ | Parallel input (i.e. $B$ or $N$) |
| $d_i =$ | Addition of $X$ deferred to this iteration. |
| $d_{i+1} =$ | Deferred addition of $X$ until next iteration. |
| $a_i =$ | Serial input (i.e. $A$ or $Q$) with index $i$. |
| $a_{i+1} =$ | Serial input (i.e. $A$ or $Q$)with index $i+1$. |

Recoding is done using the following constraints.

It may not be obvious but there are really two streams of Booth encoding for $A$ and $Q$, one for even numbers and one for odd numbers. Therefore it is necessary to ensure that the deferred addition of $B$ or $N$ is added to the correct $R$ calculation. An improvement on this design can be seen in [23] where the number of iterations was reduced by using modified Booth recoding but this design suffers from slightly more complex selection logic (Table 5).

Finally Fig. 8 illustrates minimizing the loop bound using look-ahead.

The primary disadvantage with implementing the interleaved architecture is that adding negative values by inverting a number and adding 1 to the carry input requires that the carry input value after dividing by 4 should be zero. Achieving this requires using an additional 2 carry save adders to ensure the carry has rippled past the first two carry bits.

To implement the interleaved approach without negative values, the carry input value needs to be determined by carryin = floor $((4*c_1+2*c_0+2*s_1+s_0)/4)$. The values of $c_1, c_0, s_1,$ and $s_0$ are dependent on the previous carry save number and the value of $(2*q_{i+1}+q_i)N$. $s_0' = s_0 \oplus n_0$, $c_0' = s_0 n_0$, $s_1' = s_1 \oplus c_0 \oplus n_1$, $c_1' = s_1 c_0$ or $s_1 n_1$ or $c_0 n_1$. In order to ensure that the carryin value is less than or equal to 1 then $c_1' c_0' s_1'$ must equal zero. This requires that $(s_1 c_0$ or $s_1 n_1$ or $c_0 n_1)(s_0 n_0)(s_1 \oplus c_0 \oplus n_1)$. If $n_0$ is zero (as in the case of 2N) then this constraint is satisfied. If $n_1$ is zero then the constraint becomes $(s_1 c_0)(s_0 n_0)(s_1 \oplus c_0) = s_1 c_0 s_0 n_0 (s_1 \oplus c_0) = 0$ since both $s_1$ and $c_0$ must be 1 to satisfy the AND term but lead to 0 for the exclusive OR term. Therefore, if and only if $N$ is of the format where $n_1 = 0$ and $n_0 = 1$ then the carryin will be guaranteed to be less than or equal to 1. If $N$ is of the

**Table 4.**
**Recoding in interleaved Montgomery multiplier.**

| Inputs | | | Outputs | | Meaning |
|---|---|---|---|---|---|
| $d_i$ | $a_{i+1}$ | $a_i$ | $X$ | $d_{i+1}$ | |
| 0 | 0 | 0 | 0 | 0 | Plain old times 0 |
| 0 | 0 | 1 | $X$ | 0 | Plain old times 1 |
| 0 | 1 | 0 | $2X$ | 0 | Plain old times 2 |
| 0 | 1 | 1 | $-X$ | 1 | Beginning of string of 1's |
| 1 | 0 | 0 | $X$ | 0 | End of string of 1's |
| 1 | 0 | 1 | $2X$ | 0 | End of string of 1's |
| 1 | 1 | 0 | $-X$ | 1 | End one, start another string of 1's |
| 1 | 1 | 1 | 0 | 1 | Middle of string of 1's |

**Table 5.**
**Example: Interleaved Montgomery multiplier illustrated.**

| Cycle | $(2a_{i+1}+a_i)B$ | $X_i+(2a_{i+1}+a_i)B$ | $(2q_{i+1}+q_i)N$ | $X_i$ |
|---|---|---|---|---|
| 1 | $(2a_0+a_{-1})B$ | – | – | – |
| 2 | $(2a_1+a_0)B$ | $X_{-1}+(2a_0+a_{-1})B$ | $(2q_0+q_{-1})N$ | – |
| 3 | $(2a_2+a_1)B$ | $X_0+(2a_1+a_0)B$ | $(2q_1+q_0)N$ | $X_1$ |
| 4 | $(2a_3+a_2)B$ | $X_1+(2a_2+a_1)B$ | $(2q_2+q_1)N$ | $X_2$ |
| 5 | $(2a_4+a_3)B$ | $X_2+(2a_3+a_2)B$ | $(2q_3+q_2)N$ | $X_3$ |
| 6 | $(2a_5+a_4)B$ | $X_3+(2a_4+a_3)B$ | $(2q_4+q_3)N$ | $X_4$ |
| 7 | $(2a_6+a_5)B$ | $X_4+(2a_5+a_4)B$ | $(2q_5+q_4)N$ | $X_5$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| n | $(2a_{n-1}+a_{n-2})B$ | $X_{n-3}+(2a_{n-2}+a_{n-3})B$ | $(2q_{n-2}+q_{n-3})N$ | $X_{n-2}$ |
| $n+1$ | – | $X_{n-2}+(2a_{n-1}+a_{n-2})B$ | $(2q_{n-1}+q_{n-2})N$ | $X_{n-1}$ |
| $n+2$ | – | – | – | $X_n$ |

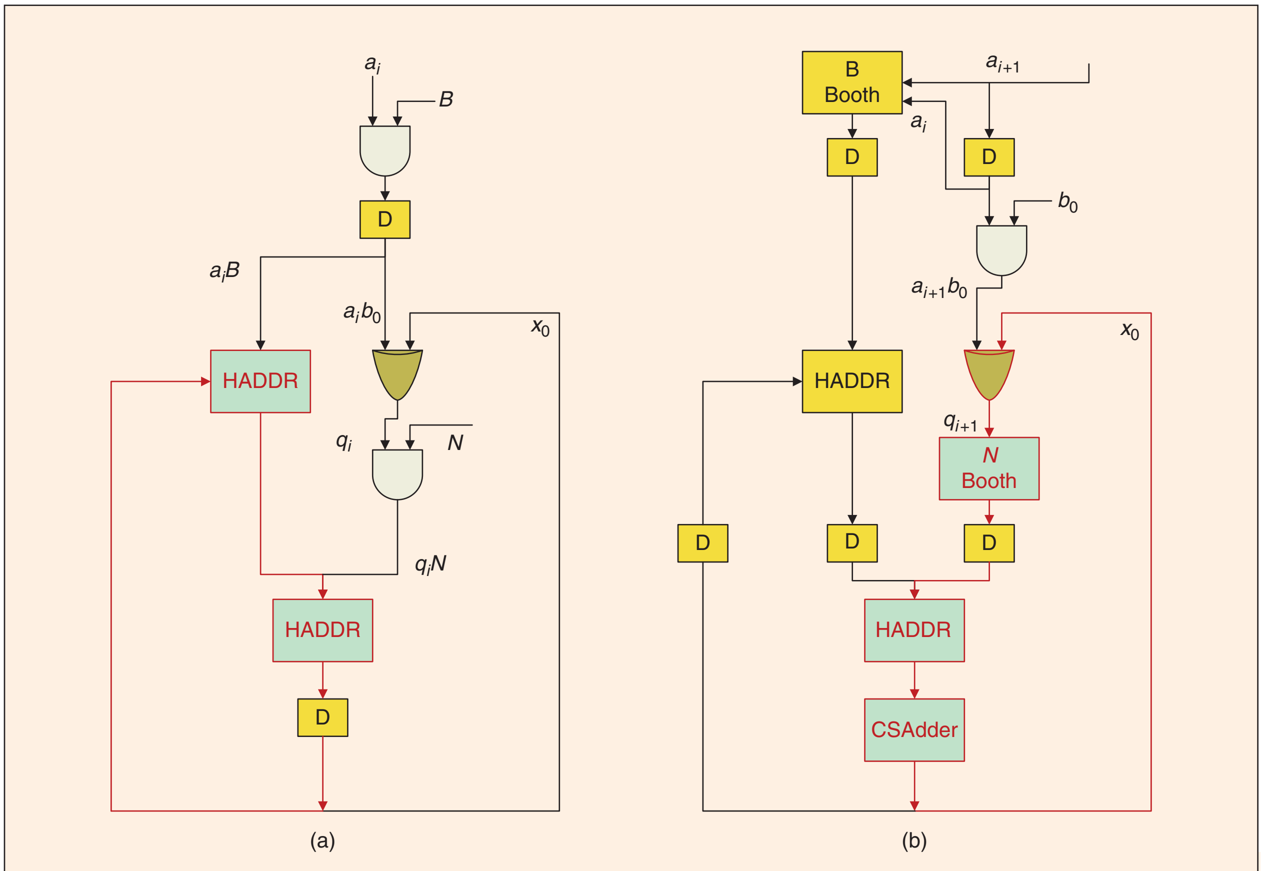


**Figure 8.** Standard (a) versus interleaved (b) loop bound.

**Table 6.**
**Interleaved synthesis results for 4-bits to 1,024-bits.**

| Bit | Slice | Slice-FF | 4 Input LUT | Bonded IOBs | GCLKS |
|---|---|---|---|---|---|
| Maximum | 10752 | 21504 | 21504 | 240 | 32 |
| 4 | 67 | 52 | 125 | 13 | 1 |
| 512 | 3787 | 3617 | 7261 | 1029 | 1 |

**Table 7.**
**Frequency performance comparison.**

| Architecture | Operand Size | Minimum Period | Maximum Frequency |
|---|---|---|---|
| Serial Squarer | 4 | 1.433 ns | 698.0332 MHz |
| Serial Squarer | 1,024 | 2.832 ns | 353.107 MHz |
| Systolic | 4 | 1.936 ns | 516.556 MHz |
| Systolic | 1,024 | 1.906 ns | 524.728 MHz |
| Alt. Systolic | 4 | 1.536 ns | 650.978 MHz |
| Alt. Systolic | 1,024 | 1.536 ns | 650.978 MHz |
| Interleaved | 4 | 3.948 ns | 253.318 MHz |
| Interleaved | 1,024 | 4.146 ns | 241.210 MHz |

format where the last 2 bits are 1 then an additional adder should be used to clear the carry bits.

The critical paths of the traditional architecture and the interleaved architecture are illustrated in red in Fig. 8. The critical path for the traditional architecture is equal to $t_{CP} = t_{\text{HADDR}} + t_{\text{HADDR}}$, where $t_x$ represent time for element $x$, CP stands for the critical path, HADDR stands for a hybrid carry save adder. This critical path is also the loop bound. The critical path for the interleaved architecture is equal to $t_{CP} = t_{\text{HADDR}} + t_{\text{CSADDER}} + t_{\text{XOR}} + t_{\text{BOOTH}}$, where $t_x$ represent time for element $x$, CP stands for the critical path, HADDR stands for a hybrid carry save adder, CSADDER stands for a carry-save adder that clears the lower carry bits, XOR stands for an exclusive-OR logic gate, and BOOTH stands for Booth recoding implementation. This critical path is also the loop bound.

Table 6 shows the synthesis results for implementing the interleaved design using 4-bit, and 512-bit cases. These results were synthesized using Xilinx ISE Webpack version 12.1 for the Xilinx Virtex xc4vlx25-12sf363. The Xilinx

**It is reasonable to assume a similar or greater speedup when implementing these RSA architectures in state of the art 45 nm technology which leads to 2,500 to 4,000 RSA operations/second.**

software has a builtin limit on the generate command which prevented us from obtaining results for 1,024-bits and higher. The synthesized minimum period for a 4 bit implementation of the systolic architecture is 3.948 ns with a maximum frequency of 253.318 MHz on the Virtex xc4vlx25-12sf363. The synthesized minimum period for a 512 bit implementation of the systolic architecture is 4.146 ns with a maximum frequency of 241.210 MHz on the Virtex xc4vlx25-12sf363.

A typical implementation of the RSA algorithm's modular exponentiation operation would require $O(\log_2(E))$ modular squaring and $O(\log_2(E))$ modular multiplication operations, where $E$ represents the exponent in the modular exponentiation. The traditional architecture uses $n$ cycles. If two traditional multipliers are used in parallel, then the total cycle complexity is approximately $\log_2(E) * (n)$. The total speed per modular exponentation then becomes $(\log_2(E) * (n)) * t_{CP}$. If the critical path is less than 9.536 ns then the RSA modular exponentiation operation will complete in less than 10 ms for $n = 1{,}024$. This leads to over 100 RSA operations per second. The interleaved architecture uses $n$ cycles. If two interleaved multipliers are used in parallel, then the total cycle complexity is approximately $\log_2(E) * (n)$. The total speed per modular exponentation then becomes $(\log_2(E) * (n)) * t_{CP}$. If the critical path is less than 9.536 ns then the RSA modular exponentiation operation will complete in less than 10 ms for $n = 1{,}024$. This leads to over 100 RSA operations per second.

## V. Comparisons

In the previous sections several different circuit architectures were derived that are used in RSA modular exponentiation implementations.

Table 7 illustrates a comparison between the architectures in terms of the synthesized critical paths. As to be expected from the designs, the frequency values do not drop significantly with increasing the operand length. Also, the minimum period does not increase significantly with increasing the operand length. The squarer architecture shows a slowdown when increasing the operand length but the synthesized results only show a register for the cause of the delay.

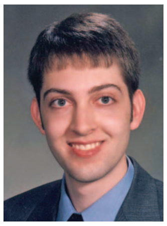
Using the numbers from Table 5, then the squarer architecture can complete one RSA modular exponentiation in 6 ms, the systolic arrays can complete 1 RSA modular exponentiation in a little over 4 ms, and the interleaved architecture can complete 1 RSA modular exponentiation in under 5 ms on our target FPGA (Virtex xc4vlx25-12sf363).

## VI. Conclusion

This paper has introduced several optimization techniques for RSA cryptography acceleration. The main methods that are described are serial squaring, systolic array implementations of Montgomery multipliers, and the interleaved architecture techniques.

On a Virtex xc4vlx25-12sf363 FPGA, the serial squarer architecture can be used to design an RSA modular exponentiation method which requires 6 ms to complete (~166 RSA/second), the systolic architectures can be used to design an RSA modular exponentiation method which requires a little over 4 ms to complete (~250 RSA/second), and the interleaved architecture can be used to design an RSA modular exponentiation method which requires less than 5 ms to complete (~200 RSA/second).

In 2009, the Intel atom core was ported to a Xilinx Virtex 5 and achieved 50 MHz performance [24]. The low end ASIC implementations run at 800 MHz leading to a speedup of 16 when going from FPGA to ASIC. It is reasonable to assume a similar or greater speedup when implementing these RSA architectures in state of the art 45 nm technology which leads to 2,500 to 4,000 RSA operations/second. Additional throughput can be achieved through parallel RSA operations.



**Aaron E. Cohen** received the B.S. degree in Computer Engineering at the University of Illinois Urbana Champaign in 2002. Afterwards, he attended the University of Minnesota Twin Cities where he received an M.S.E.E. degree in 2004 and Ph.D. in 2007.

His current research interests include security, voice over Internet protocol (VoIP), design of algorithms, VLSI architectures, and circuits for cryptography and communication systems, with emphasis on error-correcting coding, modular arithmetic, and finite field arithmetic. He is a current member of the IEEE.

In 2003, he worked part time as a Research and Development Intern for Medical Graphics Corporation, which is a medical device company located in Saint Paul, MN.

In 2005, he worked as a Research and Development Intern at Leanics Corporation on 10 Gigabit Ethernet. Beginning in 2006, he became Principal Investigator of the SBIR Phase 1 contract N06-086: Tactical Secure Voice/

Variable Data Rate Inter Working Function which focuses on developing novel solutions to secure Voice over Internet Protocol (VoIP). The Phase 2 contract began in 2008.

In 2010, he was competitively selected for an award for a new SBIR Phase 1 contract while working at Leanics Corporation.

**Keshab K. Parhi** received his B.Tech., MSEE, and Ph.D. degrees from the Indian Institute of Technology, Kharagpur, the University of Pennsylvania, Philadelphia, and the University of California at Berkeley, in 1982, 1984, and 1988, respectively. He has been with the University of Minnesota, Minneapolis, since 1988, where he is currently Distinguished McKnight University Professor in the Department of Electrical and Computer Engineering. His research addresses VLSI architecture design and implementation of physical layer aspects of broadband communications systems, error control coders and cryptography architectures, high-speed transceivers, and ultra wideband systems. He is also currently working on intelligent classification of biomedical signals and images, for applications such as seizure prediction, lung sound analysis, and diabetic retinopathy screening. He has published over 450 papers, has authored the textbook *VLSI Digital Signal Processing Systems* (Wiley, 1999) and coedited the reference book *Digital Signal Processing for Multimedia Systems* (Marcel Dekker, 1999).

Dr. Parhi is the recipient of numerous awards including the 2004 F.E. Terman award by the American Society of Engineering Education, the 2003 IEEE Kiyo Tomiyasu Technical Field Award, the 2001 IEEE W.R.G. Baker prize paper award, and a Golden Jubilee award from the IEEE Circuits and Systems Society in 1999. He has served on the editorial boards of the *IEEE Transactions on CAS, CAS-II, VLSI Systems, Signal Processing, Signal Processing Letters*, and *Signal Processing Magazine*, and served as the Editor-in-Chief of the *IEEE Trans. on Circuits and Systems- I* (2004–2005 term), and currently serves on the Editorial Board of the *Journal of VLSI Signal Processing*. He has served as technical program cochair of the 1995 IEEE VLSI Signal Processing Workshop and the 1996 ASAP conference, and as the general chair of the 2002 IEEE Workshop on Signal Processing Systems. He was a distinguished lecturer for the IEEE Circuits and Systems Society during 1996-1998. He was an elected member of the Board of Governors of the IEEE Circuits and Systems Society from 2005 to 2007.

## References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystem," *Commun. ACM*, vol. 21 no. 2, pp. 120–126, Feb. 1978.

[2] W. Diffie and M. E. Hellman. (1976). New directions in cryptography. IEEE Trans. Inform. Theory [Online]. 22(6), 644–654. Available: http://ieeexplore.ieee.org

[3] P. Kocher, "Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems," in *Proc. 16th Annu. Int. Cryptology Conf.—Advances in Cryptology (CRYPTO'96) (LNCS 1109), Santa Barbara*, CA. Berlin: Springer-Verlag, Aug. 1996, pp. 104–113.

[4] P. Kocher, J. Jaffe, and B. Jun. (1999). Differential Power Analysis [Online]. LNCS 1666, 388–397. Available: citeseer.ist.psu.edu/kocher99differential.html

[5] K. K. Parhi, *VLSI Digital Signal Processing*. New York: Wiley, 1999.

[6] S. S. Ghoreishi, M. A. Pourmina, H. Bozorgi, and M. Dousti, "High speed RSA implementation based on modified booth's technique and Montgomery's multiplication for FPGA platform," in *Proc. 2nd Int. Conf. Advances in Circuits, Electronics and Micro-Electronics*, 2009, pp. 86–93.

[7] A. Miyamoto, N. Homma, T. Aoki, and A. Satoh, "Systematic design of RSA processors based on high-radix montgomery multipliers," *IEEE Trans. VLSI*, pp. 1–11, 2010.

[8] M. Shieh, J. H. Chen, W. Lin, and H. Wu, "A new algorithm for high-speed modular multiplication design," *IEEE Trans. Circuits Syst. I*, vol. 56, no. 9, pp. 2009–2019, 2009.

[9] N. Nedjah and L. Mourelle, "Three hardware architectures for the binary modular exponentiation: Sequential, parallel, and systolic," *IEEE Trans. Circuits Syst. I*, vol. 53, no. 3, pp. 627–633, 2006.

[10] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Crypto '86 Proc. (LNCS 263)*. Berlin: Springer-Verlag, 1987, pp. 311–323.

[11] S. Lu, S. Zhang, Y. Zhang, J. Han, and X. Zeng, "Architectural integration of RSA accelerator into MIPS processor," in *Proc. IEEE 8th Int. Conf. ASIC (ASICON)*, 2009, pp. 948–951.

[12] A. P. Fournaris, "Fault and simple power attack resistant RSA using Montgomery modular multiplication," in *Proc. 2010 IEEE Int. Symp. Circuits and Systems (ISCAS)*, 2010, pp. 1875–1878.

[13] K. A. Bayam and B. Ors, "Differential power analysis resistant hardware implementation of the RSA cryptosystem," in *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, 2008, pp. 3314–3317.

[14] A. Berzati, C. Canovas, and L. Goubin, "In (security) against fault injection attacks for CRT-RSA implementations," in *Proc. 5th Workshop Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2008, pp. 101–107.

[15] J. Ha, C. Jun, J. Park, S. Moon, and C. Kim, "A new CRT-RSA scheme resistant to power analysis and fault attacks," in *Proc. 3rd Int. Conf. Convergence and Hybrid Information Technology (ICCIT)*, 2008, pp. 351–356.

[16] A. Miyamoto, N. Homma, T. Aoki, and A. Satoh, "Chosen-message SPA attacks against FPGA-based RSA hardware implementations," in *Proc. 2008 Int. Conf. Field Programmable Logic and Applications (FPL)*, 2008, pp. 35–40.

[17] M. Joye, "Protecting RSA against fault attacks: The embedding method," in *Proc. Workshop Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2009, pp. 41–45.

[18] P. A. Wang, W.-C. Tsai, and C. B. Shung, "New VLSI architectures of RSA public-key cryptosystem," in *Proc. 1997 IEEE Int. Symp. Circuits and Systems (ISCAS'97)*, June 9–12, 1997, vol. 3, pp. 2040–2043.

[19] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone. (1996). *Handbook of Applied Cryptography* [Online]. Boca Raton, FL: CRC. Available: http://www.cacr.math.uwaterloo.ca/hac/

[20] D. E. Knuth, *The Art of Computer Programming, vol. 2, Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1997.

[21] P. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[22] W. L. Freking and K. K. Parhi, "Performance-scalable array architectures for modular multiplication," *J. VLSI Signal Process.*, vol. 31, pp. 101–116, 2002.

[23] J.-J. Leu and A.-Y. Wu, "Design methodology for booth-encoded Montgomery module design for RSA cryptosystem," in *Proc. IEEE Int. Symp. Circuits and Systems*, May 28–31, 2000, pp. 357–360.

[24] P. H. Wang, J. D. Collins, C. T. Weaver, B. Kuttanna, S. Salamian, G. N. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, and H. Wang, "Intel[textregistered] atomTM processor core made FPGA-synthesizable," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays (FPGA)*, 2009, pp. 209–218.